GDB Introduction
15-213 Spring 2008

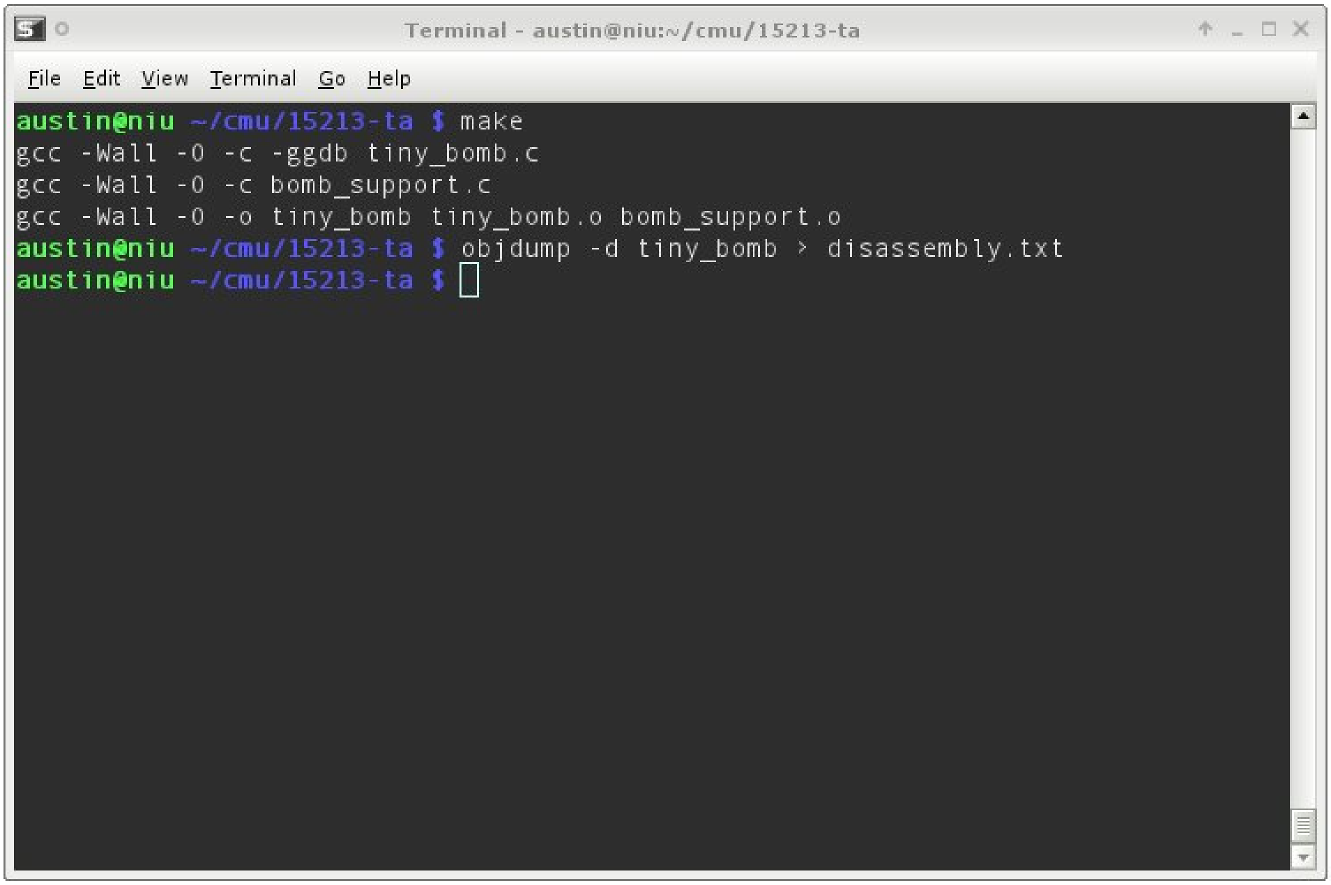
This document should serve as a useful introduction to debugging with GDB. We've put together a simple bomb with a single phase and provided some of the source code, below:

```
/* tiny bomb! this is a much smaller, simpler bomb for demonstrating
 * gdb and basic assembly language */

#include <stdio.h>
#include <stdlib.h>
#include "bomb_support.h"
#include "phase_support.h"

int main(int argc, char *argv[])
{
    /* start reading from standard in */
    FILE *infile;
    infile = stdin;
    char *input = read_line(infile);

    /* haHA! this will cut the class size in half, leaving more
     * time for my evil research! */
    if (!phase_n(input))
        fail_213_student();
    else
        phase_n_defused();

    exit(0);
}
```

This code is similar in spirit to the much more complicated multi-phase bombs you are working on for the second lab. The main function reads in an input string from STDIN and passes it to the phase, which validates it in some way, returning 1 on success and 0 on failure.

This small snippet of C is very valuable: the overall structure of the code is much clearer than it would be if we had to work this logic out by hand using the debugger.

Let's fire up our tools and get started:

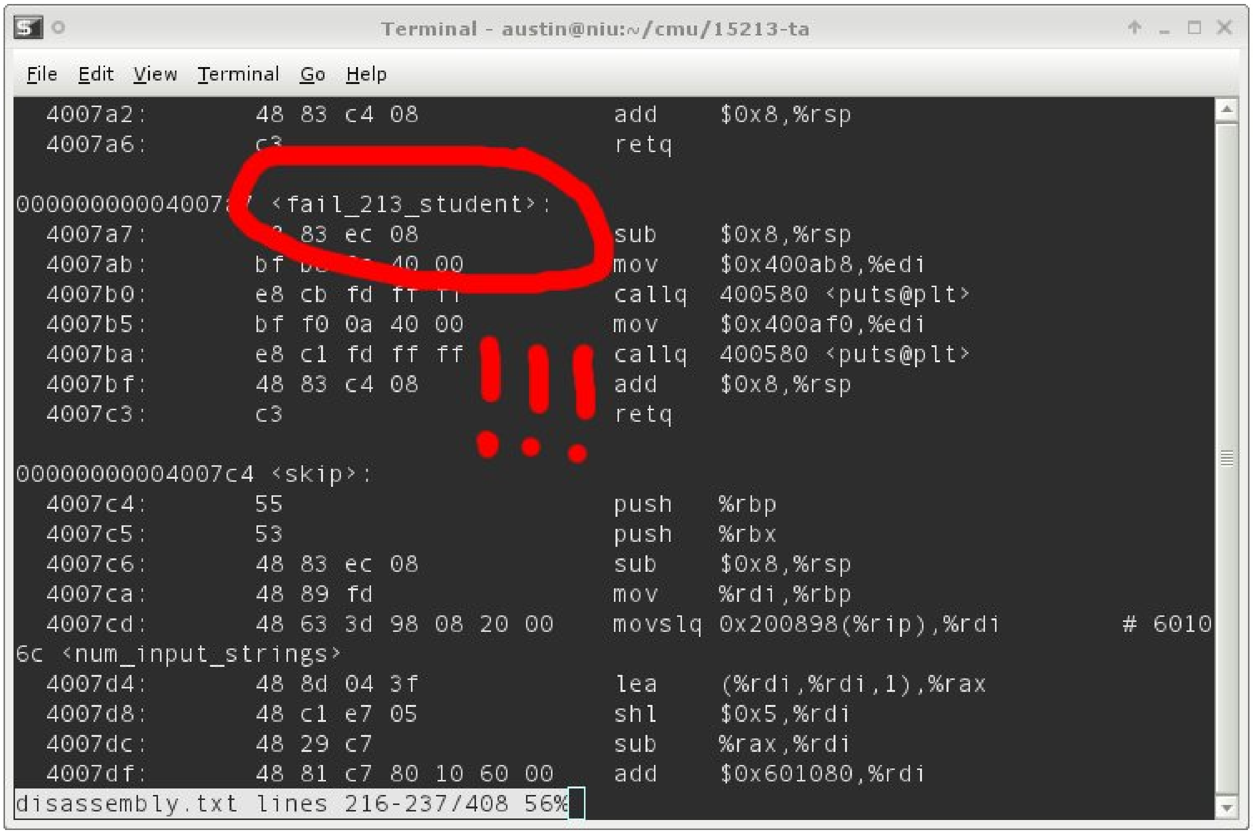


Start by producing a disassembly of the bomb

By using objdump, we can produce a disassembly of the binary Doctor Evil left us. In other words, we've gone from the original C code (that we don't have access to) down to an x86 binary (which has absolutely zero human readable information), and then back to x86 assembly (which we can at least attempt to read). It is very important to realize that this is a destructive process: we lose all kinds of information about the original program structure, like variable names, function names, statement ordering, and so on. A major component of this lab is to learn how to take such garbled code and slowly piece together the C code that originally produced it.

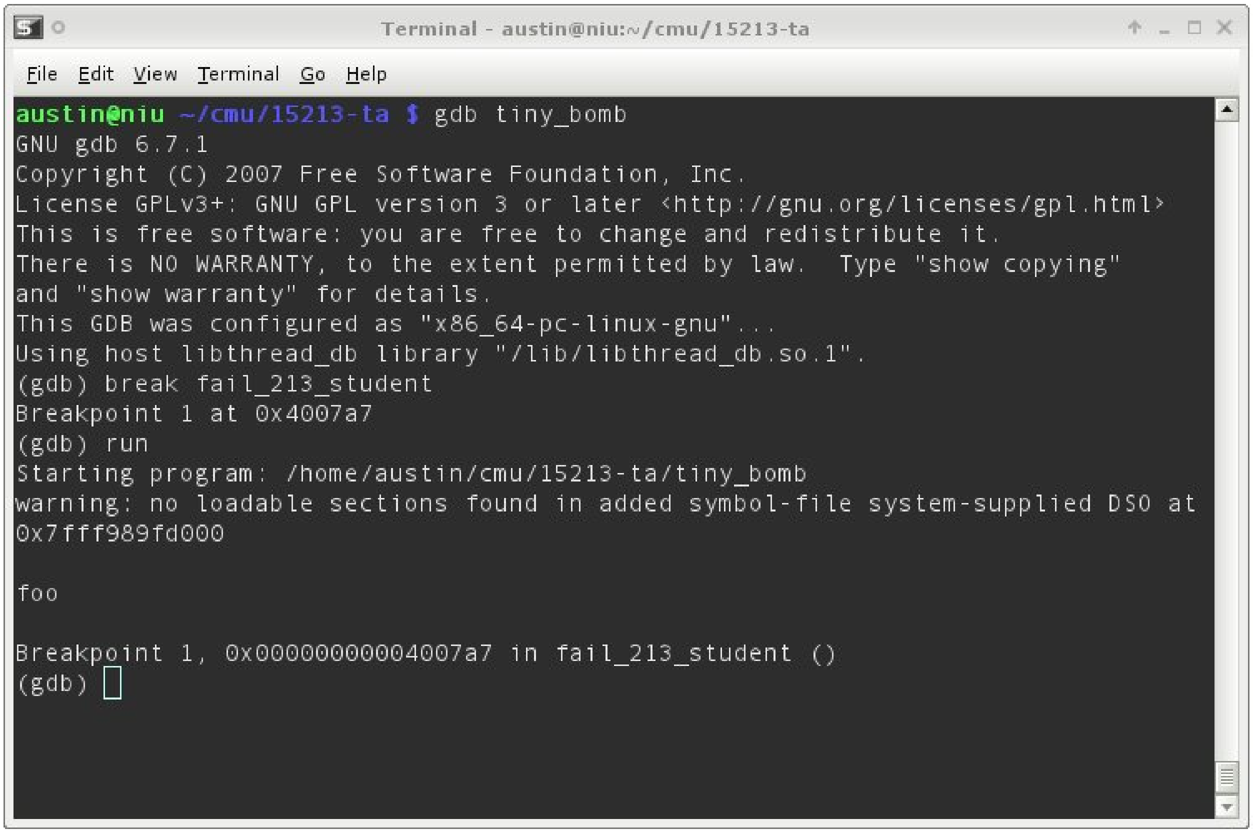
Also note that you have recieved a binary compiled with most of the debugging flags turned on and most of the optimizations turned off, giving you a tremendous advantage. In an optimized binary, it is actually very difficult even to do things like group a particular series of instructions into a single function, or identify which areas of memory are responsible for storing certain pieces of state.

Looking at our C listing again, we should immediately be concerned about the fail_213() function. It looks like Doctor Evil is playing for keeps: if that function ever manages to run, the consequences could be dire. We're going to have to find some way to tinker with this bomb without ever running that function. First though, let's take a look at the disassembly we just produced:



Scan through the disassembly, and look for anything "suspicious"

Let's be careful. If we just try and run this bomb in an uncontrolled environment, we won't have any way of stopping this function from running. Fortunately, GDB has an incredibly useful feature called "breakpoints." With a breakpoint, we can execute the bomb normally until it reaches a certain instruction (in this case, the first instruction of fail_213_student) and stop execution right there. We could then examine the state of the process, call another function, look at the contents of memory, etc. In this case, though, we're going to be most interested in stopping execution so as to not trigger the bomb. See below for a demonstration of setting a breakpoint:

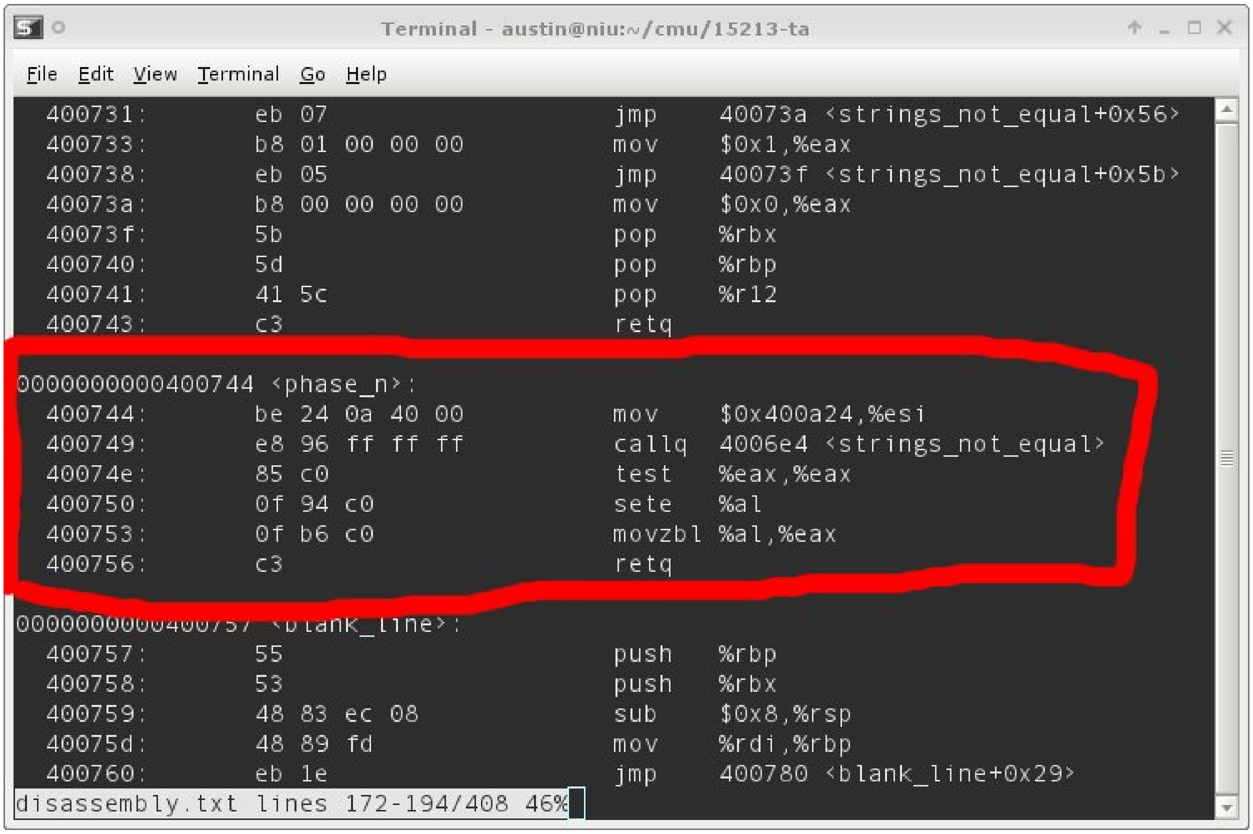


Load the bomb in GDB so we can set a breakpoint on the "dangerous" function.
Notice that now when we run the bomb, execution stops when we reach the breakpoint

As you can see, when we run the bomb with a breakpoint set, execution stops before the first instruction of fail_213_student runs, saving us from having to give an embarrasing explanation to our friends.

Don't forget that whenever you quit GDB, all your breakpoints are cleared! Be sure to set up your breakpoints whenever you start a new debugging session.

Now that we can work on the bomb without fear, let's see if we can discover what phase_n() is doing by looking at the dissassembly:



```
Terminal - austin@niu:~/cmu/15213-ta
File  Edit  View  Terminal  Go  Help
  400731:       eb 07                   jmp    40073a <strings_not_equal+0x56>
  400733:       b8 01 00 00 00          mov    $0x1,%eax
  400738:       eb 05                   jmp    40073f <strings_not_equal+0x5b>
  40073a:       b8 00 00 00 00          mov    $0x0,%eax
  40073f:       5b                      pop    %rbx
  400740:       5d                      pop    %rbp
  400741:       41 5c                   pop    %r12
  400743:       c3                      retq

0000000000400744 <phase_n>:
  400744:       be 24 0a 40 00          mov    $0x400a24,%esi
  400749:       e8 96 ff ff ff          callq  4006e4 <strings_not_equal>
  40074e:       85 c0                   test   %eax,%eax
  400750:       0f 94 c0                sete   %al
  400753:       0f b6 c0                movzbl %al,%eax
  400756:       c3                      retq

0000000000400757 <blank_line>:
  400757:       55                      push   %rbp
  400758:       53                      push   %rbx
  400759:       48 83 ec 08             sub    $0x8,%rsp
  40075d:       48 89 fd                mov    %rdi,%rbp
  400760:       eb 1e                   jmp    400780 <blank_line+0x29>
disassembly.txt lines 172-194/408 46%
```

Back to the disassembly: we can tell by looking at the C that phase_n() gets called right before fail_213_student, so let's figure out what phase_n() is doing

This looks pretty short and simple: we're moving some hex constant into a register, calling a function, and checking its return value. Without knowing the function prototypes for anything besides phase_n, we can still put together a rough outline of the function body. Your TA will go over the process of filling in the blanks below:

```
int phase_n(char *foo)
{
    int x = strings_not_equal(                ,                  )

    if (x ==          )
       return 0;
    else
       return 1;
}
```

To fill in these blanks, let's start by figuring out the function arguments to strings_not_equal(). We know that the hex constant in the first line is an argument, because %esi is the register that stores the second function argument (look at your x86_64 handouts!). Trick question: why don't we see a line that copies something into %edi, the register that stores the first argument?

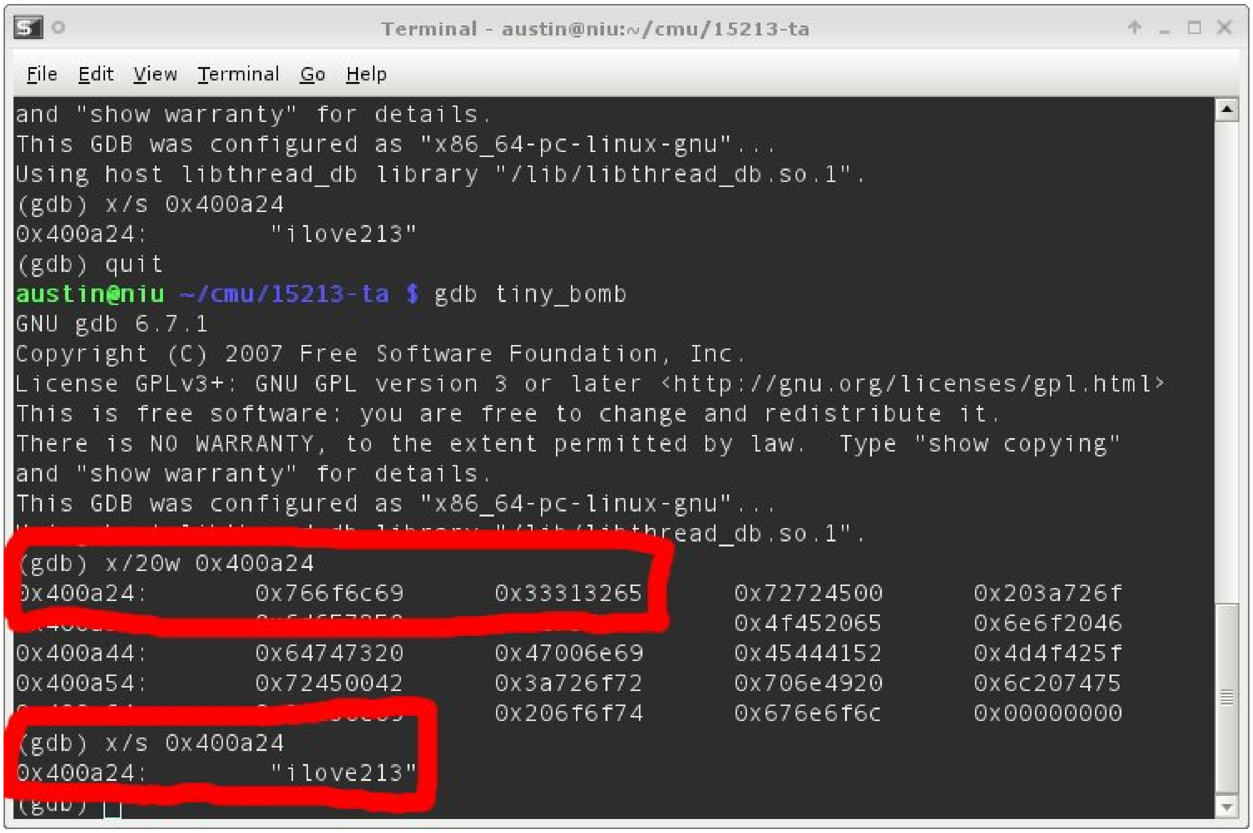
If strings_not_equal is comparing two strings (as its name suggests), we can probably assume that the hex constant being stored in %esi is some kind of pointer, probably a char *. Let's dig into it with GDB:

```
Terminal - austin@niu:~/cmu/15213-ta
File  Edit  View  Terminal  Go  Help
austin@niu ~/cmu/15213-ta $ gdb tiny_bomb
GNU gdb 6.7.1
Copyright (C) 2007 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-pc-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) x/20w 0x400a24
0x400a24:       0x766f6c69      0x33313265      0x72724500      0x203a726f
0x400a34:       0x6d657250      0x72757461      0x4f452065      0x6e6f2046
0x400a44:       0x64747320      0x47006e69      0x45444152      0x4d4f425f
0x400a54:       0x72450042      0x3a726f72      0x706e4920      0x6c207475
0x400a64:       0x20656e69      0x206f6f74      0x676e6f6c      0x00000000
(gdb) 
```

If we look at that address in memory, we can see that there sure is SOMETHING going on, but it isn't clear what we're looking at

This isn't very helpful, but we've still learned something: the data stored at that address is intialized even before the program starts, so the pointer can't point to anything on the stack or the heap. An even more accurate technique for identifying random bytes in memory is to familiarize yourself with the x86_64 memory image layout. The addresses of various segments are well-defined and useful to know.

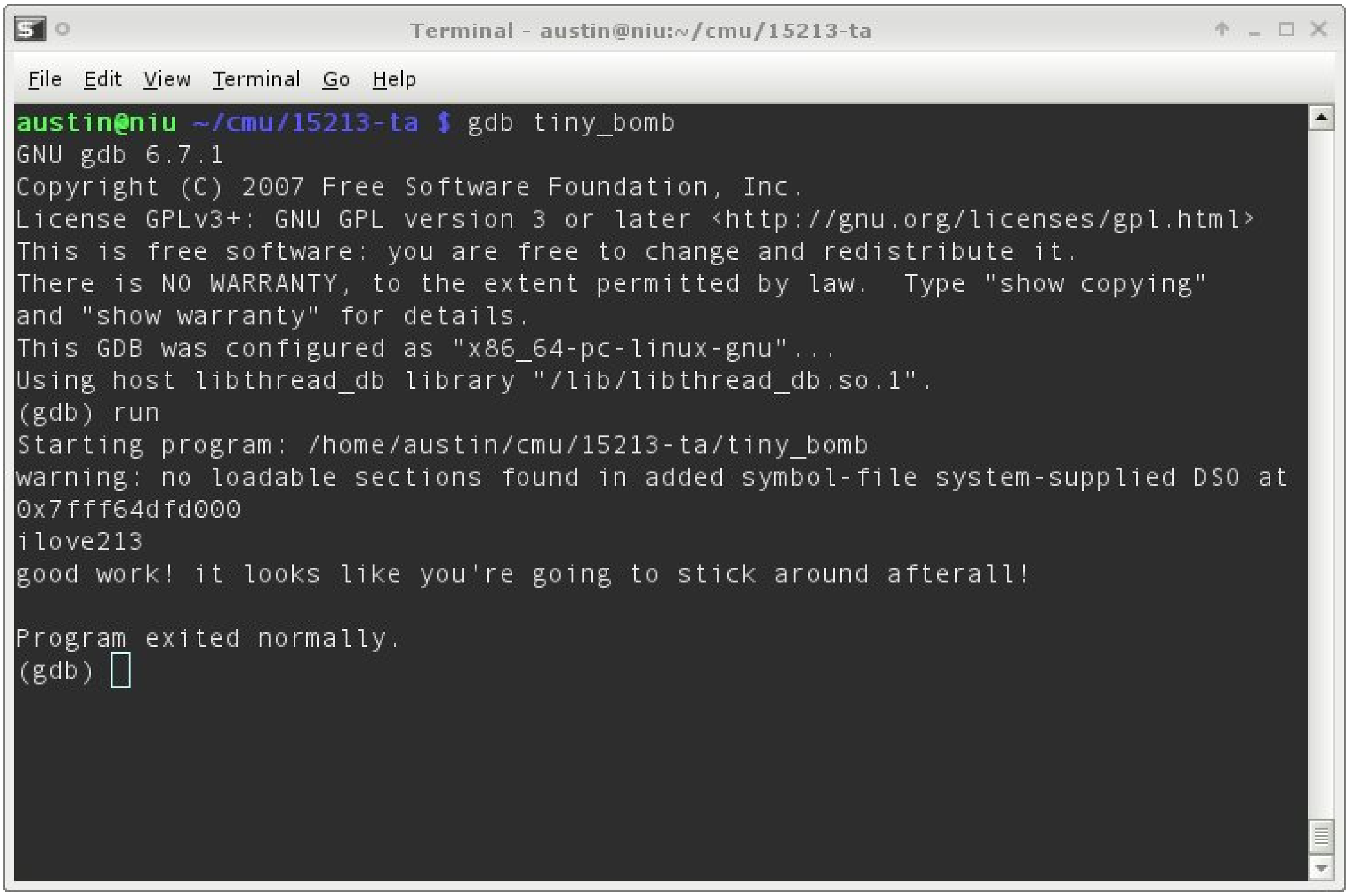
Anyways, let's make a wild guess and assume that the second argument to strings_not_equal is a char *, and that the data we're looking at is a C-style string. We can ask GDB to treat it that way by changing the flags we pass to the examine command:



AHA! If we treat the data as a C-style string, the picture gets a little clearer.

Now we're getting somewhere: it looks like phase_n is comparing a hard-coded string ("ilove213") to another string, and returning 1 or 0 based on the result of that comparison. Notice from the addresses that we're looking at the same addresses, just with a different format string. GDB has no way of knowing ahead of time that a particular series of bytes is a C-string, or an array of integers, or a floating point number. It is up to you to make those deductions!

Now that we've identified this address as a pointer to a C string, let's make another leap and assume that the two strings phase_n is passing to strings_not_equal need to be equal. If we make one final guess and assume that phase_n is expecting it's argument to be "ilove213"...



Success! Notice that we didn't have to reverse every last line of the dissassembly: with a few clever guesses and only a small amount of work, we solved the phase. For this lab, it will be important to not spend time reconstructing relatively straightforward-sounding functions, like strings_not_equal. Focus on the code for the phases themselves, and always keep a picture in your head of your best guess as to what the original C code looked like. Good luck!

For your reference, the original phase source code is reproduced below. Make sure you see how this source produced the assembly we examined earlier:

```
#include "phase_support.h"

int phase_n(char *input)
{
    if (strings_not_equal(input, "ilove213"))
        return 0;
    else
        return 1;
}

void fail_213_student()
{
    printf("WAHAHAHAHA! you failed; better luck next semester!\n");
    printf("(please learn about breakpoints)\n");
}

void phase_n_defused()
{
    printf("good work! it looks like you're going to stick around
afterall!\n");
}
```