CS 0449 – Puzzle Lab

This is a walkthrough to help you examine a sample Puzzle program like those you will be solving for your project.

Walkthrough

Log onto `thoth.cs.pitt.edu` and execute the following, replacing USERNAME with your user id:

```
cd /u/SysLab/USERNAME

cp ../shared/recitation .
```

recitation is the executable that you are attempting to solve. Let's load it into the debugger:

```
gdb recitation
```

A good place to start is to break at `main`, since I assured you that the programs were written in C only:

```
(gdb) b main
Breakpoint 1 at 0x804847d
```

And let's run the program until `main` is executed:

```
(gdb) r
Starting program: /u/SysLab/wahn/rec/recitation
Reading symbols from shared object read from target memory...(no
debugging symbols found)...done.
Loaded system supplied DSO at 0xffffe000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x0804847d in main ()
```

At this point, we can stop and disassemble the code in main to get an idea of what is going on. I've highlighted the function calls in bold because they are often a good place to start looking.

```
(gdb) disas
Dump of assembler code for function main:
0x08048212 <main+0>:    push   %ebp
0x08048213 <main+1>:    mov    %esp,%ebp
0x08048215 <main+3>:    sub    $0x98,%esp
.
. STACK INITIALIZATION OMITTED TO FIT ON PAGE
.
0x08048231 <main+31>:   movl   $0x8090a68,0xfffffff0(%ebp)
0x08048238 <main+38>:   mov    0x80a6c1c,%eax
0x0804823d <main+43>:   mov    %eax,0x8(%esp)
0x08048241 <main+47>:   movl   $0x64,0x4(%esp)
0x08048249 <main+55>:   lea    0xffffff78(%ebp),%eax
0x0804824f <main+61>:   mov    %eax,(%esp)
0x08048252 <main+64>:   call   0x8048d84 <fgets>
0x08048257 <main+69>:   lea    0xffffff78(%ebp),%eax
0x0804825d <main+75>:   mov    %eax,(%esp)
0x08048260 <main+78>:   call   0x80481d4 <chomp>
0x08048265 <main+83>:   movl   $0x0,0xfffffff4(%ebp)
0x0804826c <main+90>:   lea    0xffffff78(%ebp),%eax
0x08048272 <main+96>:   mov    %eax,(%esp)
0x08048275 <main+99>:   call   0x804fa08 <strlen>
0x0804827a <main+104>:  cmp    %eax,0xfffffff4(%ebp)
0x0804827d <main+107>:  jae    0x8048291 <main+127>
0x0804827f <main+109>:  lea    0xffffff78(%ebp),%eax
0x08048285 <main+115>:  add    0xfffffff4(%ebp),%eax
0x08048288 <main+118>:  incb   (%eax)
0x0804828a <main+120>:  lea    0xfffffff4(%ebp),%eax
0x0804828d <main+123>:  incl   (%eax)
0x0804828f <main+125>:  jmp    0x804826c <main+90>
0x08048291 <main+127>:  lea    0xffffff78(%ebp),%eax
0x08048297 <main+133>:  mov    %eax,0x4(%esp)
0x0804829b <main+137>:  mov    0xfffffff0(%ebp),%eax
0x0804829e <main+140>:  mov    %eax,(%esp)
0x080482a1 <main+143>:  call   0x804f9d4 <strcmp>
0x080482a6 <main+148>:  test   %eax,%eax
0x080482a8 <main+150>:  jne    0x80482c2 <main+176>
0x080482aa <main+152>:  lea    0xffffff78(%ebp),%eax
0x080482b0 <main+158>:  mov    %eax,0x4(%esp)
0x080482b4 <main+162>:  movl   $0x8090a70,(%esp)
0x080482bb <main+169>:  call   0x8048d6c <printf>
0x080482c0 <main+174>:  jmp    0x80482ce <main+188>
0x080482c2 <main+176>:  movl   $0x8090a9e,(%esp)
0x080482c9 <main+183>:  call   0x8048d6c <printf>
0x080482ce <main+188>:  mov    $0x0,%eax
0x080482d3 <main+193>:  leave
0x080482d4 <main+194>:  ret
```

We see `fgets`, an indication of where we are doing some input. We see a couple `printfs` that will do some output, we have `chomp`, a function that is not part of the standard library and is a mystery, and finally, we have a `strcmp`.

While it's tempting to go look at `chomp`, the fact we see a `strcmp` seems immediately more promising, since this might be doing the test against the solution. We know from our class discussions about the stack that the two pointer arguments `strcmp` expects will be set up on the stack by the code immediately before the call, so let's put a breakpoint in at the call.

```
(gdb) b *0x080482a1
Breakpoint 2 at 0x80482a1
```

When we want to put a breakpoint at an arbitrary address, we need to use a star prefixed to it. Let's continue running the program until we hit this second breakpoint.

```
(gdb) c
Continuing.
something
```

Along the way it's just going to pause, this is the `fgets` waiting for you to enter something. So let's enter "something." Now, breakpoint 2 will be hit, and we can look back at our disassembly listing (either on the prior page or by reissuing the `disas` command) to see where the data is we'd be interested in. We see that the contents of EAX were moved onto the stack, so maybe EAX contains one of the string pointers we're interested in. We can eXamine the memory location at the address in EAX. Notice that we use the $ here instead of the % to talk about registers. I don't know why this is inconsistent from the AT&T syntax.

Also the `/s` tells examine that we want to treat this as a string. You can look in the help for x by typing "`help x`" in gdb to learn about more formats.

```
(gdb) x/s $eax
0x8090a68 <_IO_stdin_used+4>:    "bcdefg"
```

That looks promising. It's certainly not the string we input, maybe it is the solution. Let's restart the program and try this as the solution

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

We hit the first and second breakpoints, let's continue through them, entering in the string we just discovered.

```
Breakpoint 1, 0x0804821b in main ()
(gdb) c
Continuing.
bcdefg

Breakpoint 2, 0x080482a1 in main ()
(gdb) c
Continuing.
Sorry! Not correct!

Program exited normally.
```

Hmm, that wasn't it. Let's go back and take a look at the other argument to `strcmp`. First let's disable the breakpoint at `strcmp`:

```
(gdb) disable 2
```

And let's put one in at the other argument being set.

```
(gdb) b *0x08048297
Breakpoint 3 at 0x8048297
(gdb) r
```

Restart, continue at the `main` breakpoint (1), and try entering "`bcdefg`" again. Now at breakpoint 2, we want to see what is at the address being pointed to by EAX, so let's use examine. This time, `x/s` will get confused, so we need to tell it that $EAX contains a `char` pointer by doing a cast. In general, gdb will accept C syntax in terms of variables, arrays, and memory references.

```
(gdb) x/s (char *)$eax
0xffffd5f0:      "cdefgh"
```

Hmm, we didn't enter "`cdefgh`" we entered "`bcdefg`" – the program must have altered the string somehow.

Your mission

Figure out what happened to the input string, and discover the string you need to enter in order to unlock the program.

If you have trouble, ask the TA for a hint. Show him the solution when you are done.