CS 0449 – gdb

This is a walkthrough to help you examine a sample program and figure out the password the program expects. This will help establish the basic usage of the gdb debugger, which lets you inspect programs as they are running!

Walkthrough

Log onto `thoth.cs.pitt.edu` and do your work on that machine. Use the provided command from the course website to download the puzzle executable we will be working with in the worksheet.

"puzzle" is the executable that you are attempting to solve. Once you have downloaded it to your machine and it is in the current directory, let's load it into the debugger:

```
gdb puzzle
```

A good place to start is to break at `main`, since we can assume here that this is a rather normal C program: (**b** – sets a **B**reakpoint)

```
(gdb) b main
Breakpoint 1 at 0x11a6
```

And let's run the program until `main` is executed: (**r** – **R**uns the executable) It will stop here because of our established breakpoint. (**NOTE**: the addresses may be different than the ones you see here.)

```
(gdb) r
Starting program: /afs/pitt.edu/home/d/w/dww9/puzzle

Breakpoint 1, 0x00005555555551a6 in main ()
(gdb) |
```

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. They give us great information about the high-level behavior.

```
(gdb) disas
Dump of assembler code for function main:
   0x00005555555551a2 <+0>:     push   %rbp
   0x00005555555551a3 <+1>:     mov    %rsp,%rbp
=> 0x00005555555551a6 <+4>:     sub    $0x10,%rsp
   0x00005555555551aa <+8>:     lea    0xe53(%rip),%rdi
   0x00005555555551b1 <+15>:    mov    $0x0,%eax
   0x00005555555551b6 <+20>:    callq  0x555555555050 <printf@plt>
   0x00005555555551bb <+25>:    mov    0x2e9e(%rip),%rax
   0x00005555555551c2 <+32>:    mov    %rax,%rdx
   0x00005555555551c5 <+35>:    mov    $0x64,%esi
   0x00005555555551ca <+40>:    lea    0x2eaf(%rip),%rdi
   0x00005555555551d1 <+47>:    callq  0x555555555060 <fgets@plt>
   0x00005555555551d6 <+52>:    lea    0x2ea3(%rip),%rdi
   0x00005555555551dd <+59>:    callq  0x555555555179 <chomp>
   0x00005555555551e2 <+64>:    lea    0x2e97(%rip),%rax
   0x00005555555551e9 <+71>:    mov    %rax,-0x8(%rbp)
   0x00005555555551ed <+75>:    jmp    0x555555555206 <main+100>
   0x00005555555551ef <+77>:    mov    -0x8(%rbp),%rax
   0x00005555555551f3 <+81>:    movzbl (%rax),%eax
   0x00005555555551f6 <+84>:    add    $0x1,%eax
   0x00005555555551f9 <+87>:    mov    %eax,%edx
   0x00005555555551fb <+89>:    mov    -0x8(%rbp),%rax
   0x00005555555551ff <+93>:    mov    %dl,(%rax)
   0x0000555555555201 <+95>:    addq   $0x1,-0x8(%rbp)
   0x0000555555555206 <+100>:   mov    -0x8(%rbp),%rax
   0x000055555555520a <+104>:   movzbl (%rax),%eax
   0x000055555555520d <+107>:   test   %al,%al
   0x000055555555520f <+109>:   jne    0x5555555551ef <main+77>
   0x0000555555555211 <+111>:   lea    0x2e68(%rip),%rsi
   0x0000555555555218 <+118>:   lea    0xdfd(%rip),%rdi
   0x000055555555521f <+125>:   callq  0x555555555070 <strcmp@plt>
   0x0000555555555224 <+130>:   test   %eax,%eax
   0x0000555555555226 <+132>:   jne    0x555555555236 <main+148>
   0x0000555555555228 <+134>:   lea    0xdf4(%rip),%rdi
   0x000055555555522f <+141>:   callq  0x555555555030 <puts@plt>
   0x0000555555555234 <+146>:   jmp    0x555555555242 <main+160>
   0x0000555555555236 <+148>:   lea    0xdf5(%rip),%rdi
   0x000055555555523d <+155>:   callq  0x555555555030 <puts@plt>
   0x0000555555555242 <+160>:   mov    $0x0,%eax
   0x0000555555555247 <+165>:   leaveq
   0x0000555555555248 <+166>:   retq
```

This is the output of the “disas” (**disas**semble) command that shows you the assembly code at the current position of the program as it is being executed.

We see `fgets`, an indication of where we are doing some input. We see a couple `printf`s 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 **C ABI** that the two pointer arguments `strcmp` expects will be set up by the code immediately before the call by using the conventional registers `RDI` and `RSI`, so let's put a breakpoint just before the call happens. (See the underlined section in the assembly code in the previous section for where this address came from. You can also say: `b *(main + 125)`)

```
(gdb) b *0x000055555555521f
Breakpoint 2 at 0x000055555555521f
```

When we want to put a breakpoint at an arbitrary address, we need to use a star prefixed to it. Let's continue (**c** – **C**ontinues execution) 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 `RDI` are being calculated as an address, and this register is the first argument according to the ABI, so maybe `RDI` contains one of the string pointers we're interested in. We can examine the memory location at the address in `RDI`. 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. (**x** – eXamine memory)

Also the `/s` tells the examine command 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 $rdi
0x55555555601c: "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. (**r** – restarts if the program is running)

```
(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, 0x00005555555551a6 in main ()
(gdb) c
Continuing.
Enter in the password: bcdefg

Breakpoint 2, 0x000055555555521f in main ()
(gdb) c
Continuing.
ACCESS DENIED!!

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 `main` to more easily get to the breakpoint we care about:

```
(gdb) disable 1
```

Restart, try entering "`bcdefg`" again. Now at breakpoint 2, we want to see what is at the address being pointed to by RSI, so let's use examine. Sometimes `x/s` will get confused, so we can always tell it that RSI contains a `char` pointer by doing a cast. In general, gdb will accept C syntax in terms of variables, arrays, and memory references. It will even accept many C-style expressions.

```
Breakpoint 2, 0x000055555555521f in main ()
(gdb) x/s (char*)$rsi
0x555555558080 <str>:   "cdefgh"
```

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

Your turn!

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