Project Traps

Due: October 2, 2019 at 11:59pm

1 Introduction

Alas! Tom finally pried open the door to Atlantis after successfully solving the puzzles. But he soon finds himself in dangerous waters. Some evil C-creatures have set traps in Atlantis for any visitors. The undersea traps need to be disarmed, and the only way to do that is to give each creature a specific password. Help Tom disarm them quickly to get the treasure and make his escape!

There are four guardian sea creatures, each of which will require a specific phrase to disarm the trap. Each phase expects you to type a particular string on stdin. If you type the correct input, then that creature deactivates its trap and you can proceed to the next one. Otherwise, the sea creature will know you are trying to steal the treasure and they will activate the trap, causing your terminal to temporarily lock you in (NOTE: All it really does is temporarily block off signals (which you will learn about in a few weeks), and sleep for a few seconds.) and then terminate. You then start at your respective hiding place and can try again. You can only get the treasure when all four defenses have been deactivated.

There are too many traps for one person to deal with, so we are giving each student their own traps to deactivate alone (see *Section 2.1: Collaboration*). Your mission, which you have no choice but to accept, is to deactivate your traps before the due date. Good luck, and welcome to the trap squad!

2 Assignment

Your job for this project is to deactivate your trap. Doing so will require you to employ all of your knowledge of the x86-64 assembly language. Sea creatures are smart and their traps are not easy to beat.

In this assignment, you will be providing input to a pre-compiled binary executable file. Do not run your trap on any computer outside of the computer science department. These binary files were compiled to run on those machines and there is no guarantee that they will run correctly on a different machine. Using `ssh` or FastX to log in remotely to run and deactivate your traps is fine.

2.1 Collaboration

As with Project Data, the nature of this assignment is quite different from the others in this course; consequently, the normal CS33 collaboration policy will be modified for this assignment.

For this assignment:

- You may not share any aspect of your trap, including the binary or disassembled code, for any part of this assignment with other students in this course.
- You may not discuss the solutions to any part of this assignment with other students, even at a high level.
- You may not assist other students with deactivating their traps, or look at any other student's binary or disassembled code. Only you or a member of the course staff may examine any aspect of your code.
- You may not search for solutions to these phases on the internet.

2.2 Hours Policy

In hours, TAs will not be looking at any code generated by the traps executable. TAs can only answer questions about GDB and general assembly questions, i.e. “What does this command do?”.

2.3 Obtaining Your Trap

You can obtain your trap by running the command

```
cs0330_install traps
```

This script will copy the following file into your cs0330/course/traps directory:

- *traps*: A symbolic link[1] to the executable binary trap

2.4 Deactivating Your Trap

You can (and should!) use many tools to help you deactivate your trap. Please look at section 3 (the *Hints* section) for some tips and ideas. The best way is to use your favorite debugger (cough cough **`gdb`**) to step through the disassembled binary, frequently toggling breakpoints as you go.

Each trap consists of four levels, with each level more difficult than the one preceding it. Although levels get progressively harder to deactivate, the expertise you gain as you move from level to level should offset this difficulty. Nevertheless, you should *certainly* avoid waiting until the last minute to start deactivating your trap.

Some of the levels call various support functions within the trap. **These functions will not open your trap.** The only functions that can cause the trap to pop open are the **`level()`** functions. Additionally, support routines with a name indicating a behavior do indeed exhibit that behavior - for example, a function named **`read_three_characters()`** does indeed read three characters. You will need to decipher the return values and side effects of such functions, but you will not need to scan them in detail to avoid popping the trap.

The trap ignores blank input lines. Additionally, the trap accepts input from files. If you run your trap with a command line argument, for example,

```
./traps solution.txt
```

then it will read the input lines from *solution.txt* until it reaches EOF (end of file), and then switch over to `stdin`.

3 Hints

There are many ways of deactivating your trap. You can disassemble the trap, examine it in great detail, and figure out exactly what it does without ever running it. This is a useful technique, but is not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to deactivate it. In other words, step through key functions and examine the registers while doing so. This should be enough to figure out what input will not call `pop_trap`. This is probably the fastest way of deactivating it, and is our recommended approach.

[1] You can think of a symbolic (or “soft”) link as a file or folder which acts as a pointer to another item in the filesystem. The Traps installation script creates a file in your course directory with a symbolic link to one master executable in the cs33 course directory. Any interactions with your local link will be carried out using the master executable instead. This saves you disk space, as you do not have to retain a copy of the executable.

However, *do not use brute force!* You could write a program that will try every possible key to find the right one. But this is no good for several reasons:

- Every time you guess wrong, a message is saved in the filesystem. You could very quickly saturate the filesystem with these messages and cause the system administrators to revoke your computer access.
- You have no way of knowing (without examining the trap) how long the passwords are, or what characters are in them (the passwords can contain any ASCII character). Even if you made the senseless assumptions that they all are less than 80 characters long and only contain letters, then you will have 26^{80} guesses for each phase. This will take a very long time to run (without even accounting for the delay caused by popping the trap) and you will not get the answer before the assignment is due.

There are many tools designed to help you figure out both how programs work and what is wrong when programs don't work. Here is a list of some of the tools you may find useful in analyzing your trap, and hints on how to use them.

- `gdb`

  The GNU debugger, **`gdb`**, is a command line debugger tool available on virtually every platform. You can trace through a program line by line, examine memory and registers, look at both the source code[2] and assembly code, set breakpoints, set memory watchpoints, and write scripts.

  There are many **`gdb`** resources available to you, including the `gdb` Cheatsheet located on the course website. The CS:APP web site, http://csapp.cs.cmu.edu/public/students.html, has a very handy single-page `gdb` summary that you can print out and use as a reference. Here are some other tips for using `gdb`:

  - **Toggle breakpoints liberally.** It will be very difficult for you to decipher a phase without being able to evaluate the state of the program during execution. Setting frequent breakpoints, and `print`ing different parts of the program state while stopped, will help you understand what a phase is doing throughout its execution and save you a lot of time. Note that because the start address of the program may change each time you GDB the program, you should rely on offsets, which never change. For example, GDB can break on function `level_one`, and you can also break on command `level_one`+4 if a command happens to start there. The command at level_one+4 will stay the same regardless of the precise start location of the program.
  - Use **`print`** `and` **`x/s`** frequently. This command will allow you to both examine the state of the trap, and examine data stored within the trap that you will need to deactivate it. In particular, if you see a memory access to an absolute address,

[2] Although not in this assignment.

try `printing` the data stored at that address in different formats so you can determine what it is. You can print a string this way with the command **`print (char *) <address>`**. The **`print`** statement in **`gdb`** can cast and dereference values the same way you would when writing C. You can also use **`x/s`** to see the string stored in a register.

Printing in assembly can be difficult as you are looking at variables of different data types (strings, ints, etc). For guidance on how to examine specific values, check out the gear up! These links on using the print and x commands in gdb might also be helpful!

- **`layout asm`** and similar commands are helpful tools to look at code as you go. They will make it a lot easier to step through the machine code line by line (use **`si`** and **`ni`**).
- For online documentation, type "`help`" at the `gdb` command prompt, or type "`man gdb`" or "`info gdb`" at a Unix prompt. Some people also like to run `gdb` under `gdb-mode` in `emacs`.
- You can cast values and dereference pointers in `gdb` with the C operators `(<type>)` and `*` respectively. You can glean a lot more information about the program state by using these operations in `gdb print` statements, as described above.
- You can use the `disassemble` command to disassemble a function within `gdb`, displaying its instructions and object code. It also displays the *offsets* for each instruction from the start of the function, helpful for setting breakpoints.
- You can use the `source` command to pass in a command file. This will run all the commands, line separated, in the specified file, which might be useful when deactivating later stages. You must be careful though, since a typo in the command file might cause your trap to open.

- `objdump -t traps`

  Used outside of **`gdb`**, this will print out the trap's symbol table. The symbol table includes the names of all functions and global variables in the trap, the names of all the functions the trap calls, and their addresses. This is a good place to start, as it can help you see the overall structure of the program. Looking at the function names may help you figure out where to put breakpoints!

- `objdump -d traps`

Use this to disassemble all of the code in the trap. You can also just look at individual functions. Reading the assembler code can tell you how the trap works.

Although `objdump -d` gives you a lot of information, it doesn't tell you the whole story. Calls to system-level functions are displayed in a cryptic form. For example, a call to `sscanf()` might appear as:

```
8048c36:  e8 99 fc ff ff  call   80488d4 <_init+0x1a0>
```

To determine that the call was to `sscanf()`, you would need to disassemble within `gdb` using the `disassemble` command.

For most convenient use, send the output of `objdump` to a file and annotate that file, keeping it open alongside you as you work. You can also use **layout asm** in gdb to examine the block of assembly code as you walk through.

- `strings traps`

  This utility will display the printable strings (sequences of at least 4 consecutive printable characters) in your trap.

If you are looking for a particular tool or for documentation, the commands `apropos`, `man` and `info` are your friends. In particular, `man ascii` might come in handy. `info gas` will give you more than you ever wanted to know about the GNU Assembler.

Lastly, some other tips:

- Break each function down into parts, and annotate those parts with what they do. You shouldn't try to understand each instruction individually, but instead groups of instructions.
- Write out in prose what the jump instructions do. Jump instructions in `x86_64` are not usually intuitive, so it will help you to think of how they work in a different way.

4 Grading

Your grade for this assignment is determined by the number of levels you successfully deactivate.

See the table below for guaranteed grade cutoffs. If you do not meet the threshold for a given letter grade, you may still receive that grade after Professor Doeppner applies a curve (you will only ever be curved up).

Grade	Guaranteed Grade Requirements

A	Solve all 4 levels
B	Solve the first 3 levels
C	Solve the first 2 levels
D	Solve the first level

To clarify the role of setting off traps as part of your grade: we do keep a log of the times it opens, and will use this to detect brute forcing and other such nasty techniques. However, there is no hard and fast “limit” on the number of pops you may have, nor will we explicitly take off points for pops. Don't abuse this policy. And don't sweat the few times you forget to set breakpoints in gdb.

4.1 Late Policy

If you solve additional levels after the due date, you should know that this will count as handing in again, with the corresponding late deduction applied. Normal late policy applies to this project.

Because of this, for traps, there is no way to push an NC with some functionality up to a C before the next project deadline - Since there is no way for you, as the student, to make attempts and it is an autograded project.

5 Handing In

There is no explicit handin for this project, as the trap continuously updates your score. NOTE that your score will not update if you disarm a trap from within GDB; you must disarm each level outside of gdb as well, for it to be recorded. You can check your current progress by running

```
cs0330_traps_progress
```

from the terminal.