← Project 3: What's the Password?

Due by midnight, Saturday 11/3 (or late on Sunday)

(project concept, portions of the writeup, and your generated executables thanks to Dr. Misurda.)

A great way to learn how something works is to **break it.** This is especially true for things where you don't know *how* they work. Many of the things we know about biology, the universe, physics etc. come from seeing what happens when things go wrong.

The goal of this project is to **recover passwords - or password patterns - for three executable files.** You **don't have the source code** to these programs, only the compiled executable!

To make it more of a challenge, the programs *may* have a single, fixed password, or they *may* use some kind of algorithm to generate/check the password.

> Read this section before doing anything else!

Here's how to set up `gdb` to use the Intel disassembly syntax. While logged into thoth, do this:

```
pico ~/.gdbinit
```

Inside that file, write this exactly:

```
set disassembly-flavor intel
```

and save. Now, when you view disassembly in `gdb`, it will match the slides I gave you and will be way easier to understand overall.

[40 points] `mystrings.c`

There are many tools that will be helpful for password-cracking. One is a program to extract **readable strings** from a binary file.

A binary file can contain text, but it will be embedded within a bunch of non-text data. Here's a snippet of an example output of the `hexdump -C` command, which shows the bytes in hex on the left, and their ASCII version (or . if it's not ASCII) on the right.

```
...
000002e0  21 00 00 00 12 00 00 00  00 00 00 00 00 00 00 00  |!
000002f0  00 00 00 00 00 00 00 00  00 5f 5f 67 6d 6f 6e 5f  |.
00000300  73 74 61 72 74 5f 5f 00  6c 69 62 63 2e 73 6f 2e  |s
00000310  36 00 70 72 69 6e 74 66  00 71 73 6f 72 74 00 5f  |6
00000320  5f 6c 69 62 63 5f 73 74  61 72 74 5f 6d 61 69 6e  |_
00000330  00 47 4c 49 42 43 5f 32  2e 32 2e 35 00 00 00 00  |.
00000340  02 00 00 00 02 00 02 00  01 00 01 00 10 00 00 00  |.
00000350  10 00 00 00 00 00 00 00  75 1a 69 09 00 00 02 00  |.
00000360  39 00 00 00 00 00 00 00  e8 09 60 00 00 00 00 00  |9
00000370  06 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |.
00000380  08 0a 60 00 00 00 00 00  07 00 00 00 01 00 00 00  |.
00000390  00 00 00 00 00 00 00 00  10 0a 60 00 00 00 00 00  |.
000003a0  07 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |.
000003b0  18 0a 60 00 00 00 00 00  07 00 00 00 04 00 00 00  |.
000003c0  00 00 00 00 00 00 00 00  48 83 ec 08 e8 7b 00 00  |.
000003d0  00 e8 0a 01 00 00 e8 d5  02 00 00 48 83 c4 08 c3  |.
000003e0  ff 35 12 06 20 00 ff 25  14 06 20 00 0f 1f 40 00  |.
000003f0  ff 25 12 06 20 00 68 00  00 00 00 e9 e0 ff ff ff  |.
...
```

There is a UNIX program called `strings` which lets you search for strings within a binary file like this. You will write a little program called `mystrings` which will be a very simplified version of `strings`.

How it should work

Your program will be **very** simple. Seriously. This can be done in about 30-40 lines of C, not counting braces.

`mystrings` will work as follows:

- You will run it like `./mystrings somefile`.
- It will read `somefile` as a **binary** file.

    - It should check if the file doesn't exist, by seeing if `fopen` returned `NULL`. If so, print an error and quit.
- It should look for sequences of 4 or more **printable ASCII characters.**
    - It should print each of those sequences on its own line.
    - A **printable ASCII character** is a `char` whose value is in the range 32 to 126, **inclusive.** So [32, 126].

That's it.

Requirements and Tips

You **don't have to support** any of the extra options that the `strings` program supports.

You **don't have to display tab and newline characters as embedded within strings, like `strings` does.**

The strings are **not necessarily 0-terminated!** They are terminated by *any* non-printable ASCII character. So, you can't really use `printf("%s")`.

Make sure your program can handle strings that are **arbitrarily long.** Since you have no idea how long a string is when you start reading it, that means you can't really allocate space to store the whole string.

> Fortunately, you only have to ***print*** ASCII characters until you hit the end. How many characters do you really have to *store?* Do you really need to store the string at all??

Try having a look at the functions in the C standard library, such as in `stdio.h` and `ctypes.h`. Maybe you'll find something useful!

Testing it

Do not run it on text files. That's not what it's for. Run it on binary files! Examples include executables, object files, image files, and more. There are lots of executables in `/bin`, such as `/bin/ls`.

You can see what your program is *supposed* to output for a file by comparing its output to the output of the real `strings` program with the following shell script:

```
#! /bin/sh
```

```
# Trust me.
strings --all $1 | sed 's/\t/\n/g' | grep -P '.{4,}' > strings
./mystrings $1 > mystrings.out
git diff --color strings.out mystrings.out
```

Now you can run it in the same directory as `mystrings` like:

```
./whatever_you_named_that_script.sh /bin/ls
```

If it prints nothing, then the files are identical!

The `>` `strings.out` is redirection - it sends the output of the program to the file `strings.out` instead of to the console.

Green lines mean your program is finding *too many* strings.

Red lines mean your program is *missing* strings.

[60 points] Password-cracking

I have generated **three executable files for each of you.**

Log into thoth. Then `cd /u/SysLab/yourusername`, like mine is `/u/SysLab/jfb42`. (The capitalization in `SysLab` is important.) There are your executables.

> The `/u/SysLab/` directory is physically stored on thoth. You cannot access it from any other computer, and any files you put in it will disappear at the end of the term when you lose access to thoth. It's a better idea to do all your work in your AFS space.

So, **copy the programs to your private directory** by doing `cp * ~/private`. From there, you can do whatever you want with em.

Your goal

When you run these executables, they wait for you to type something and hit enter. You must type the correct password to "unlock" them. The program will tell you whether or not you succeeded.

OH MY GOD HOW DO I EVEN START

Reverse engineering is like a puzzle. You have to **start with what you know.** Here are some things to know:

- ***A program may have a *different password every time you run it!*** Make sure to test it several times, on different dates, from different computers etc.
- All the programs are **written in C.**
  - The assembly you look at was generated by an algorithm, so there is *structure* to it.
- Each program will have a different password **per student,** but *how to find it* will be the same for everyone.
- All passwords will be **printable ASCII characters** and be **far less than 100 characters in length.**
  - That being said, brute-forcing will probably not be productive.

Here is a page with a lot more info that might be helpful for you.

The writeup

You will be typing up a short document which shows, for each of the three programs:

1. the **password**
   - it might be a **single fixed string**
   - or it might be **generated by an algorithm,** in which case you must **describe the algorithm**
2. an explanation of **how you found it**
   - you don't have to describe the *entire* process, but…
   - briefly describe any **failed attempts or ideas**
   - and then describe **how you succeeded.**
   - **If you couldn't find the password,** you can still get full credit for this part of the writeup by explaining your attempts and your theories on what the password is.

This is not a writing course. Please don't write a novel. There is no page count to hit. It only has to be between 1 and 2 pages, line spacing 1.5, 12pt font, 1 inch margins.

Please keep it to-the-point and technical. You don't need an intro and body and conclusion. Just write **clearly and tersely.** Technical writing is about *clarity*. (Humor and wit are still welcome, of course.)

Grading rubric

- **[5] Submission**
  - Any deviation from the submission instructions will lose you **all 5 points.**
- **[35]** `mystrings.c`
  - **[5]** Compiles and runs according to the specified interface
    - (should work with `gcc -Wall -Werror --std=c99`)
  - **[10]** follows the correct definition of a "string"
    - (at least 4 printable ASCII characters)
  - **[10]** handles strings of **arbitrary length**
    - (not just "100 characters")
  - **[5]** output is **displayed properly**
    - (one string per line, no blank lines, no debugging info etc.)
  - **[5]** Style
    - (this is a tiny program but try to make it readable)
- **[60] Passwords**
  - **[10]** Found password 1
    - If a program uses an algorithm for the password, but you only found *one*, it's half credit.
  - **[10]** Found password 2
  - **[10]** Found password 3
  - **[10]** Writeup for password 1
    - remember, you can still get full writeup credit if you didn't find the password, as long as you describe what you tried.
  - **[10]** Writeup for password 2
  - **[10]** Writeup for password 3

Submission

Follow this checklist for a free 5 points:

You should submit a gzipped tarball containing exactly **three files:**

- `mystrings.c`
  - Your username and full name should be in the comments at the top.
  - ***Never turn in a program that doesn't compile.***
- Your `mystrings` executable
- A **single** Word or PDF file named `<username>.docx/pdf` (like `jfb42.pdf`) that contains:
  - Your username and full name at the top

- The three passwords (if you found them!)
- The three program writeups

Name your file `username_proj3.tar.gz` like `jfb42_proj3.tar.gz`.

Now you can **submit as usual.**

© 2016-2018 Jarrett Billingsley