Below are the puzzles that lead to the meeting link. This is supposed to be *fun* (at least, in a nerd's perspective), so you're encouraged to attempt the hardest problem that you can. Both problems lead to the link, so you do not need to attempt both (but you're welcome to!) In case you do not have access to a *nix machine, I have set up an Ubuntu VM with everything you need:

IP address: `45.79.151.17`
Username: `guest`
Password: `password`

For the second problem, a check program is set up in the home directory, which can be invoked as
`$ ./check <answer>`
This program will check if your answer is within 0.1 of the expected answer (the solution to either of the two parts of Problem 2 will work).

Of course, there are... *other* ways to get the meeting link; but if you have the technical expertise to use those methods, feel free to! By design, Problem 1 is not set up in a particularly secure manner–again, this is supposed to be *fun*!

Important: While this VM is set up so you can use it as you wish, please be mindful of other puzzle-solvers who might be using it. In particular:

1. Please do not leave your solutions in the home directory and ruin it for everyone else. Avoid writing to files. The problem is designed to be fully solvable using pipes.
2. You're not explicitly *disallowed* from trolling others (by, for instance, swapping out the message), but the SHA256 digest might make this a bit less fruitful.
3. You are, of course, free to `scp` the problem files over.

Problem Set

1. Use the `message.txt` file and any clues in the email to figure out the meeting link.

   [*Hard mode:* Do this on Windows.]

2. Let $f(x) = \frac{1}{2}x^2$ with its domain restricted to $\mathbb{R}_+$. Let there be four points initially located at $(0, 1)$, $(1, 0)$, $(0, -1)$, and $(-1, 0)$. A *windmill* process begins at time $t = 0$, where the points rotate clockwise at infinite speed around their center[1]. As time progresses, the center of rotation moves along the curve of $f(t)$, at a speed proportional to its gradient. The four points maintain their relative distances and positions with respect to this moving center.

   As they move, the rotating points sweep out an area in the Euclidean plane. At $t = \pi$, the rotation stops instantaneously. Calculate the total area covered by the points during their rotation from $t = 0$ to $t = \pi$.

   [*Harder:* Now suppose the distance of each of the points from the center of rotation at time t is given by $f'(t)$. Calculate the new total area[2].]

   [*Something to think about:* What if f was not Riemann-integrable? For example, consider the Dirichlet function $f(x) = \mathbb{1}_{\mathbb{Q}}(x)$, which is not Riemann-integrable, but has a Lebesgue integral of 0. Since the function is everywhere-discontinuous and therefore not differentiable, assume that the windmill process in this case "teleports" to each location instead. Is the original problem well-defined for this case, and if so, what is the new area? One could argue that since $\mathbb{Q}$ is dense in $\mathbb{R}$, this area should be $2\pi^2$ (the area of the circle times the Lebesgue measure of the path, multiplied by 2 to account for both cases of the function); but the Lebesgue measure of (any subset of) $\mathbb{Q}$ is 0, so one could also argue the area should be 0. What specific part(s) of this problem make this a challenge?]

[1] More explicitly, the points cover the full circle instantaneously at every instant.
[2] Obviously, since f has a Lipschitz-continuous gradient, this problem is still well-defined. My fascination with the Hessian is widely-known at this point.

Solutions: The *intended* way

1. It is obvious that the message is encrypted; the original email failed to mention that it was with AES-256-CBC (which I later communicated). The hint in the email strongly implied that the public key that Proton uses to let you encrypt emails was the same one you should use to *decrypt* the message. From a cryptography perspective, this is strange; but this is a puzzle. The other red herring here is that the key is a PGP key, but me giving you an algorithm implies that PGP was not actually used to encrypt the message.

   The remaining solution is to use `openssl` to decrypt it. This is now quite straightforward:

   ```
   $ openssl enc -d -kfile <key file> -in encrypted.txt -aes-256-cbc 2>/dev/null
   ```

   Aside: You were not expected to know how to use `openssl`; you were expected to know that it exists, and how to use `--help`. As a nice side-effect, all the above weirdness made this rather gippity-proof!

   This gives you the following: `aHR0cHM6Ly9yeWVkaWRhLm1lL2p1c3Rhd2Vic2l0ZQo=`. From the structure of the string, and the ending = sign, it's very obvious that this is a base64-encoded string; as such, we can just use

   ```
   $ base64 -d
   ```

   to decode this. For the first time, we see a human-readable message: a URL `https://ryedida.me/justawebsite`. This is a simple page that has the text, *"Yay, you made it! You know what to do next."*. It's obvious that you're supposed to open up the console–which is empty. Look at the source instead, which is a very bare HTML page: with a stunningly opaque script tag. This single, 110k character long line, when uncommented, is actually valid JavaScript! It's written in an esoteric version called JSFuck. At this point, you have two options after uncommenting the line: copy it into the browser, or use Node.

   This will give you the following:

```
#pragma GCC optimize("O0")
#include <stdio.h>

#define TITLE       "You're on your own, kid"
#define HINT1       "Search deep within for the answers"
#define HINT2       "All good things take time"

#define N(a)        "%"#a"$hhn"
#define O(a,b)      "%45$"#a"d"N(b)
#define o(a)        "%100$"#a"d"
#define B(a,b,w)    o(*(a))o(*(b))o(w)N(a)
#define SS          "\n\033[2J\n%26$s";

char* fmt = O(37,2)O(78,3)/*__________________*/O(141,4)O(77,6)O(28,7)O(251,8)O(10,
            9)O(251,10)O/* | \    /|\    / |*/(254,11)O(257,12)O(12,13)O(255,14)O(
            141,1)O(109,/* |       |       |*/15)O(248,16)O(256,17)O(15,18)O(186,
            19)O(60,20)O/* |      /|\      |*/(255,21)O(11,22)O(186, 23)O(69,24)O(
            246,25)O(198,/*| -     o     - |*/26)O(74,27)O(236,28)O(255,29)O(261,
            30)O(251,31)O/*|                 |*/(253,32)O(209,33)O(258,34)O(204,1)N(
            1)B(6,35,25)B/*| /     |     \ |*/(7,36,25)B(8,37,25)B(9,38,25)B(10,39,
            24)B(11,40,24)/*----------------*/B(12,41,24)B(13,42,24)B(14,43,24)O(2,
            1)O(103,30)O(251,31);

#define arg d+68,d,d+1,d+2,(scanf(d,d+58),d+68),\
    d+31,d+32,d+33,d+34,d+35,\
    d+36,d+37,d+38,d+39,d+10,\
    d+11,d+12,d+13,d+14,d+15,\
    d+16,d+17,d+18,d+19,d+20,\
    d+21,d+22,d+23,d+24,d+25,\
```

```
    d+26,d+27,d+28,d+29,d+30,\
    d+58,d+59,d+60,d+61,d+62,\
    d+63,d+64,d+65,d+66,d+67

volatile char d[69] __attribute__((aligned(64))) = {37,115, 0};

int main() {
    printf(fmt, arg);
}
```

It's obvious this is C code, but it is obfuscated. Indeed, this code draws inspiration from Carlini's legendary 2020 IOCCC-winning submission[3] that relies on libc abuse. We defer to their explanation for interested readers.

By nature, the code above is meant to be hard to read and rely on undefined behavior (UB). Here is an annotated version:

```
#pragma GCC optimize("O0")
#include <stdio.h>
#include <string.h>
#include <assert.h>

// To arg a, write the number of bytes written so far mod 256
#define N(a)        "%"#a"$hhn"
// Print arg 45 with width a, then write the number of bytes written so far mod 256
// to arg b
#define O(a,b)      "%45$"#a"d"N(b)
#define o(a)        "%100$"#a"d"  // Print arg 100 with width a
// My attempt at a hard-coded memcmp. Write a and b bytes, then write a
// pre-specified width to get the number of bytes written so far to 0 (mod 256).
// Write out the number of bytes so far to a. If a and b were the same,
// 0 is written to a
#define B(a,b,w)    o(*(a))o(*(b))o(w)N(a)
// ASCII escape sequence to clear the screen, then print argument 26 as a string
#define SS          "\n\033[2J\n%26$s";

char* fmt = O(37,2) // Write 37 (%) to arg 2 (which is d)
    O(78,3)  // Write 115 (s) to arg 2 (d+1), since we already wrote 37 bytes
    O(141, 4)  // Write 0 (256 % 256) to arg 3 (d+2)
    // Write "Midnights" to d[32-41]
    O(77,6)O(28,7)O(251,8)O(10,9)O(251,10)O(254,11)O(257,12)O(12,13)O(255,14)
    O(141,1)  // Reset counter to 0
    // Write out meet.jit.si/yedida24
    O(109,15)O(248,16)O(256,17)O(15,18)O(186,19)O(60,20)O(255,21)O(11,22)
    O(186,23)O(69,24)O(246,25)O(198,26)O(74,27)O(236,28)O(255,29)O(261,30)
    O(251,31)O(253,32)O(209,33)O(258,34)
    O(204,1)  // Reset counter to 0
    N(1)
    // Attempting a memcmp, it kinda works
    B(6,35,25)B(7,36,25)B(8,37,25)B(9,38,25)B(10,39,24)B(11,40,24)
    B(12,41,24)B(13,42,24)B(14,43,24)
    O(2,1)  // Reset counter to 0
    O(103,30)O(251,31);
```

[3] https://www.ioccc.org/2020/carlini/index.html

```c
// Memory layout
// d[0-2]: %s
// d[10-30]: meet.jit.si/yedida24
// d[31-40]: Midnights (expected)
// d[58-67]: Input
// d[68]: Temporary stuff

#define arg d+68,d,d+1,d+2,(scanf(d,d+58),d+68),\
    d+31,d+32,d+33,d+34,d+35,\
    d+36,d+37,d+38,d+39,d+10,\
    d+11,d+12,d+13,d+14,d+15,\
    d+16,d+17,d+18,d+19,d+20,\
    d+21,d+22,d+23,d+24,d+25,\
    d+26,d+27,d+28,d+29,d+30,\
    d+58,d+59,d+60,d+61,d+62,\
    d+63,d+64,d+65,d+66,d+67

// volatile: don't optimize away
// aligned(64): align to 64 bytes
volatile char d[69] __attribute__((aligned(64))) = {37,115, 0};

int main() {
    printf(fmt, arg);
}
```

Aside: I'm not actually sure the `#pragma` and the alignment and `volatile` are needed, but I wanted to be as sure as I could that this worked, especially since this is well into UB territory.

That's cool, but it doesn't tell us what to do, and running the code takes input and then....does nothing except print some garbage. This is where the hints come in. The first hint (which is more of an Easter egg) is in `TITLE`, and the ASCII art gives an additional hint. The ASCII art shows a clock at midnight, and "*You're on your own, kid*" is a song from Taylor Swift's *Midnights*.

The two actual hints are clues on what to do. The first one says, "*Search deep within for the answers*". This is a message that asks you to simply print out each element of the array `d`. In testing this puzzle, I used the following:

```c
    for (int i = 0; i < 69; i++) {
        printf("d[%2d] = %c\n", i, d[i]);
    }
```

Add this at the end of `main`. You'll see the structure of `d`, where you'll see your input as well as *Midnights* (from the Easter egg). But you will also see the solution, the URL.

Where does the second hint, "*All good things take time*" come in? That was simply to let you know that the code taking a long time to run (on my machine, this took about 2-3 minutes!) was intentional, and that you should just wait for it to finish. Although you *can* use `printf` to do anything, it doesn't mean you *should*–and this is an example of why.

Aside: Why the call to `scanf`? Believe it or not, that wasn't meant to throw you off. Initially, this was supposed to be a puzzle where you used the `TITLE` and ASCII art to put in a "password", and that would give you the URL. And although `printf` is Turing-complete, which means I could have, in theory checked the answer using that (which, to my credit–I did manage to get mostly working), the undefined behavior kept screwing up the URL, which was the more important bit. I did also briefly experiment with using `memcpy` and `memcmp`, but felt that it somewhat defeated the shock of a pure PDP (*printf*-driven programming). At this point, I was about 8-9 hours into writing this piece of C code[4], so I called it. In

[4]Yes, this entire puzzle, which should've been a fun little weekend thing, became *days* of dealing with undefined behavior and random bugs. I probably spent about 12 hours in total on the puzzles. Isn't that the quintessential dev experience?

retrospect, it would've been a lot funnier if I had put the result of scanf right before the URL in the array, so putting in the wrong password could cause you to overflow and see garbage. I never actually thought of this at the time, sadly.

2. The naive approach just integrates the area of a circle times the arc length over the interval. This gives you

$$A = \int_0^\pi \pi\sqrt{1 + f'^2(x)}dx = \int_0^\pi \pi\sqrt{1 + x^2}\mathrm{d}x$$

For completeness, the solution is below, but note that technically, this solution is wrong (but since this was my first attempt, this is accepted by the program). We use the standard method to integrate $\sec^3(u)$[5].

$$\begin{aligned}
A &= \pi \int_0^\pi \sqrt{1 + x^2}dx \\
&= \pi \int_0^{\tan^{-1}\pi} \sec^3(u)\ du && \text{(substitute } x = \tan u\text{)} \\
&= \frac{\pi}{2}\left(\sec(u)\tan(u) + \ln(\sec(u) + \tan(u))\right)\Big|_0^{\tan^{-1}\pi} && \text{(see footnote)} \\
&\approx 19.195
\end{aligned}$$

The harder part would've followed this method, just adding an x^2 in the integral:

$$\begin{aligned}
A &= \pi \int_0^\pi x^2\sqrt{1 + x^2}dx \\
&= \pi \int_0^{\tan^{-1}\pi} \tan^2(u)\sec^3(u)du && \text{(substitute } x = \tan(u)\text{)} \\
&= \pi \int_0^{\tan^{-1}\pi} (\sec^2(u) - 1)\sec^3(u)du \\
&= \pi \int_0^{\tan^{-1}\pi} \sec^5(u) - \sec^3(u)du \\
&= \frac{\pi}{4}\tan(u)\sec^3(u)\Big|_0^{\tan^{-1}\pi} - \frac{\pi}{4}\int_0^{\tan^{-1}\pi} \sec^3(u)du && \text{(using the reduction formula)} \\
&= 0.0796 - \frac{\pi}{8}\left(\sec(u)\tan(u) + \ln(\sec(u) + \tan(u))\right)\Big|_0^{\tan^{-1}\pi} \\
&\approx 83.623
\end{aligned}$$

However, both of these fail to account for the fact that the areas overlap. In fact, accounting for these gives us a far simpler integral to solve. We just need to remember that at $t = 0$ and at $t = \pi$, half the circle's area needs to be added back (for a total additional area of π):

[5] https://personal.math.ubc.ca/~feldman/m121/secx.pdf

$$
\begin{aligned}
A &= \pi + \int_0^{\pi} \left|\frac{1}{2}x^2 + 1\right| - \left|\frac{1}{2}x^2 - 1\right| dx \\
&= \pi + \int_0^{\sqrt{2}} x^2 dx + \int_{\sqrt{2}}^{\pi} 2dx \\
&= \pi + \frac{x^3}{3}\Bigg|_0^{\sqrt{2}} + 2x\Bigg|_{\sqrt{2}}^{\pi} \\
&= 3\pi - \frac{\sqrt{2}}{3} \approx 8.953
\end{aligned}
$$

Solutions: The *other* ways

I did mention that there are other ways to solve this: here are a couple of them:

1. Look closely at how Problem 2 is set up: once you solve it, you'd have to `ssh` into the machine and use the `check` program to get the URL. This means that this program has the URL! Once you make this realization, the hack is obvious:

   ```
   $ strings ./check
   ```

   The `strings` program prints out all the strings it can find in the input file. Running the above gave you the final URL directly.

2. A somewhat less straightforward alternative approach was to at least check my domain for URLs that were suspicious. The problem here is that my website uses client-side routing, so you'd have to use something like Puppeteer to scrape links. A little tedious, but it would have paid off if you did try.

3. A very easy cheat would've been to wait a bit until people solved the problem and just

   ```
   $ cat ~/.bash_history
   ```

   I fixed this by adding `unset HISTFILE` to the `.bashrc` file–which you could've simply removed and then waited.