# Cyber Matters

## 8. Conclusions

I think if I were to try to identify the main thing I'm taking away from this project, it would be a sense of general awe of and admiration for people who develop web applications as a career, as well as people who make a living out of technical cybersecurity. Web development is *hard*, and as a beginner in the field, a lot of the actual work of this project for me was reading documentation, watching tutorial videos, and patching together my code from examples online. Of course, this isn't ideal development practice, and it's even further from ideal security practice. In fact, I took some shortcuts to make this thing workable that, while effective in getting my point across, introduced major vulnerabilities that would make my code completely unviable in a production environment. Let me give you a quick example of what I mean.

```
random_year = random.randint(2012, 2019)
random_day = random.randint(1, 28)
random_month = random.randint(1, 12)
random_date = datetime.datetime(random_year, random_month, random_day)
```

This is a quick snippet from my random tweet generation code. I didn't want attackers to be able to figure out which tweets were real from the times at which they were tweeted, so I generated my own tweet times. However, I quickly ran into an issue. Can you spot it? Check out the `random_day` variable. The issue is that different months have different numbers of days. If you generate a random date with a random month and a random day in the range `[1,31]`, you'll generate some dates which have invalid numbers of days, which will quickly alert anyone looking at the date carefully that it's not real. Instead, I decided to hack it together by only generating days in the range `[1, 28]`, but this introduces its own problem: namely, that any tweet with a day greater than 28 must be a real tweet! I decided to err on that side because hopefully, the pattern won't be too obvious, but a better implementation would take this into account and generate the month first, then generate the number of days based on the month. That's just a small example of the difficulty of doing even something as seemingly easy as creating a random date, and how a small imperfection in seemingly unimportant code can create a security flaw.

A much larger problem with my application comes from the following code:

```
const path = 'http://localhost:5000/tweet_auth';
      axios.get(path)
        .then((res) => {
          if (res.data.error === 'true') {
            this.invalid_user = true;
          } else {
            this.questions = res.data.questions;
            this.question = this.questions[this.question_dex].possibles;
            this.correct = this.questions[this.question_dex].correct;
          }
        })
```

Can you spot this one? I'll give you a hint – what do you think `this.correct` is?

I went to great lengths to explain the client-server model earlier in this blog, and one of its advantages, as I mentioned, is that it provides greater security by splitting up the user and the data, and then creating strict rules about how the two can interact. However, it's not always an easy paradigm to maintain. My code above breaks one of the cardinal rules of client-server programming: never bring unnecessary data from the server up to the client. However, that's exactly what I do in my code above when I get `this.correct`, which is the value of the correct answer for the security question, from the server and store it in the client. That's kind of like if a sphinx had to go around carrying a slip of paper with the answer to its riddle on it. A novice attacker with some basic tools could inspect the DOM, the set of objects stored in your browser when you look at a web page, and get access to this correct answer.

Why did I allow this heartbreaking security flaw? Simple – I couldn't get my app to work otherwise. And I think this illuminates a really compelling point about how computer science works. The point is that it's pretty much impossible to design a system that can't be broken by incompetence. If I knew vue.js better, I would have been able to create a perfect client-server split where no exploitable information ever leaks to attackers, but I don't! Since I'm limited by the time I have this semester, I wasn't able to learn all I needed to in order to avoid this kind of mistake. Now hopefully people who spend all their time doing these kinds of things will have the requisite time and experience to avoid these mistakes, but no matter how well-intentioned a person is, a tough deadline can make them cut corners. Not to mention, of course, the classic problem: it's difficult to know what it is that you don't know. When "what you don't know" is that a tool you're working with has a terrible security flaw, it's difficult to protect yourself or design around it. Whew.

## The Takeaway

If there's one thing I want you to come away from this blog with, it's this: cyber-security is tough stuff. As my example has shown, it's the little things that get you. You can have as awesome an authentication schema as you want, but something as simple as a random date can burn you if the right attacker looks in the right place. I find this a little bit encouraging, though, because it implies that we can always get better at security. The knowledge that we can never really become perfectly secure is also a motivation to keep on working and improving. I'm proud of the work I've done in conjunction with this blog, but I also know it's riddled with flaws and places to improve. Without trying to sound too fatalistic, that's just the nature of designing systems for us flawed human beings.

Thanks for reading,

Liam Pulsifer