CSci 5271 Fall 2021 Exercise Set 4 answers template

Please use this document as a template for submitting your answers to exercise set 4. (This template is available from the course web site in either LaTeX or Google Doc formats). Type your answers on each page after the question prompt (you can use additional pages, though that we expect that would rarely be required). If you can write all your answers electronically, please do so and export to a PDF to submit. If you would prefer to hand-draw figures, you can also submit a scan.

Please ensure that the names and UMN email addresses of all of your group members are recorded on Gradescope, and also confirm them below:

Name	UMN email address
Ben Smith	smit9524@umn.edu

1. **Random numbers with limited entropy**
    a. This is a horrible idea because not only is the current seed freely available, every past and future seed is also available. Since time progresses linearly, we can always calculate the number of seconds from Jan 1, 1970 to any time since then, or in the future, giving us access to the seed of any random number generation call that the system has ever (or will ever) do/done.
    b. The range of 2 to 65535 is only 65533. Thus it would only take 65533 guesses to determine the seed. Additionally, once a seed is determined, they would be able to reasonably determine the next seed (PID) that they may get from subsequent logins. Even if there are dozens/hundreds of other users logging in at approximately the same time, if they determine that on one login the PID is 5000 (for example), then they have automatically ruled out PIDs less than 5000 for subsequent logins, and will be able to determine a new seed with even less computation.

c. By logging in once every second, a malicious user could in theory obtain an identical seed. Consider the scenario in which only one user is accessing the system at a given time, and that user logs in once every second. The time increases by one every time, and so does the PID. If we XOR (t, p) where t and p are arbitrary values for the time and the PID, we will obtain some number n. If the user logs in the next second (and no one else has logged in since they last did) we now have XOR(t+1, p+1). The XOR of these values will produce the same number n as XOR(t, p), and the user would have the same seed as when they previously logged in.

d. Yes, it would help if the function could take in some sort of fixed secret key. However, the operation with this fixed key would have to be "one-way" in and of itself, in addition to the entire seed generation function being "one-way." This prevents an attacker from determining the seed by way of accessing the secret key. A better function that includes time since the epoch, the PID, and a secret key may be to bit shift the time by PID mod the secret key, bit shift the secret key by the least significant digits of the time, and then xor these values.

```
seed = XOR( time << (PID % key), key << (time // 10) )
```

2. Cross-site scripting variations

A simple change would be to add a field to the HTTP header that specifically says the site expects no inline JS. We can do this by adding a line to the code that sets the content security policy in the HTTP header. The site will now know that it has no inline JS, and any injected JS into the username field will be ignored.

We can simply add the following line under the `resp.SetContentType` call:

```
resp.GetWriter("Content-Security-Policy: default-src https:")
```

The above line will disable any inline JS or stray eval calls.

3. **Denial of Service Denial**

Sly's scheme is not very smart. If the server is simply reinserting requests back to the end of the queue, he is still allowing a DoS attack in at least two ways.

Consider the first scenario in which a malicious user (i.e. from the same IP address) is making enough requests to fill up a significant portion of the queue. The server will spend all of its time sorting through these requests and requeuing them to handle any real requests. This may not completely deny service to normal users, but it will at least significantly slow down their response time from the server.

Consider the second scenario in which (a) malicious user(s) is making a substantial amount of requests to the server from different IP addresses (for example, a botnet). These requests may all be handled since they are coming from different IP addresses, and the server would spend all of its time processing these requests that it would be unable to process requests from normal users.

4. **Virus Virii**

a. No, Sam's claim (sadly) does not conflict with the undecidability of the halting problem. Turing and others have proved that it is impossible to write a computer program that determines the execution of another program with 100% accuracy. A simple program that does exactly what Sam claims it does would output VIRUS!!! for every executable inputted into it. Thus, it would indeed output VIRUS!!! for every virus that was passed to it, as Sam claims, but it would also output VIRUS!!! for every other executable passed in that is not a virus, which would not be very helpful.

b. A program may have 10,000 "safe" instructions, and have the 10,001st instruction and onward propagate a virus. In this way, Sam's ViruSniff would not output VIRUS!!! yet the executable would indeed propagate a virus. Another alternative would be to have the executable deposit some malicious code (ie. virus) in some part of the machine to lie dormant - depending on the signature matching algorithm this may or may not be detected, since in terms of memory usage this may look like a normal disk system write.

Sam's claim is thus incorrect. His neural network based signature matching program cannot be 100% effective, as we have just shown. Additionally, a *true* virus detection program would indeed conflict with the undecidability of the halting problem: no computer program can determine with 100% accuracy the execution of every other computer program.