Lab 07 –Python / SenseHat (4)

1 Security hardening – the continuing saga

1.1 Getting rid of Auto-login

We have previously configured the RPI such that one needs password for **sudo** events. But, some of you may have noticed that we start up with auto-login to the **pi** user in the desktop. This clearly is not a good security practice.

So, we need to ensure that login is required after the RPI starts.

```
sudo leafpad /etc/lightdm/lightdm.conf
```

In this file you'll find the setting that permits auto-login.

```
autologin-user-pi
```

You can disable this setting by simply commenting it out with the #.

```
#autologin-user-pi
```

Then save the file, and reboot.

Note:

- It is also possible to get rid of the default user suggestion.
- This is a good suggestion for a deployed production unit, but it is also quite inconvenient for everyday use. Still, it is highly recommended for production systems.
- This is left as an additional exercise for the interested student (hint: it's the same **conf** file)

1.2 Scheduling tasks

"Cron is a tool for configuring scheduled tasks on Unix systems. It is used to schedule commands or scripts to run periodically and at fixed intervals. Tasks range from backing up the user's home folders every day at midnight, to logging CPU information every hour."

More **information** at: https://www.raspberrypi.org/documentation/linux/usage/cron.md (read it)

1.2.1 What to do

a) Look over the **information** and check your default user has any scheduled tasks.
   Use `crontab -l` to do this.
b) For fun & the learning experience: install the **gnome-schedule** application.
c) Add the Hourly.py file to be run every full hour. "`python3 Hourly.py`"
d) Check that the command is run as expected.
e) Use `crontab -l` to check your entries.

The example here, to run the Hourly.py file, is of course not very important. But, it may be useful to schedule many security tasks to run at fixed times. Now you know how to do that.

f) Ok, so you installed **gnome-schedule**, did you? Well, this is not an application that we want to be present in a production environment. So, now you shall uninstall the installed **gnome-schedule** application. (Note: The cron entries you added will still be there.)

g) Reboot option – using "crontab -e"

   It is possible to schedule tasks to start when the RPI boots. This is done with the "reboot" option. The "&" is a generic way to schedule the program to be run in the background[1]. It works on any shell command you give. You don't have to use the "&" in the crontab.

```
@reboot date >> bootevent.log &
```

[1] You can use the "**`ps -l`**" command to see which processes are running. If you want, you can also kill the process by using the "**`kill <pid>`**" command. The **<pid>** is the process id (a number), and it is listed in the **ps** output.

2 A Sense Hat Event Logger - Background

We will write functions to log Sense Hat events as described in the Sense Hat API, https://pythonhosted.org/sense-hat/api/.

We differentiate between events that are **user-initiated** and events that **influence** the SenseHat. The **user-initiated** events are the joystick event. Writing to the LED matrix is an **influence** event. In addition, we will introduce a **Time** event.

2.1 JSON (JavaScript Object Notation)

We will make use of the JSON (JavaScript Object Notation) module to serialize / deserialize python objects to string format.

- https://docs.python.org/3/library/json.html#module-json
- https://realpython.com/python-json/

You don't really need to know very much about the JSON format, except that it is a very common format for communicating data objects.

We will only use the **json** module to convert lists into strings and back again. This is handled by standard **json** functionality: **json.dumps()** and **json.loads()** .

2.2 Writing to files

We will need to be able to write to text files in this exercise. The attached example file shows how to do this.

2.3 Output and the output buffer

The **print()** function does not actually print the output itself. Instead, it places the output in an output buffer for the file stdout. The stdout file is printed by the I/O system (in python this is handled by standard library function, which actually is from the C standard library).

When using **print()** just before **time.sleep(),** you may find that the printout comes after the sleeping period. If this is not what you wanted, then you may use **stdout.flush()** function to force the printout to occur.

To use **stdout** functions, you need to import the **sys** module.

2.4 User-initiated events (the Joystick)

When we previously used the **`get_event()`** function, we did not use all the functionality. In particular, we ignored the **`timestamp`**. We will now make use of the **timestamp** too.

Now, we will use all fields of **event**:

- timestamp - float
- direction – “up”, “down”, ”left”, “right”, “middle”
- action – “pressed”, “released” “held”

The timestamp is encoded in the same way as is the c standard library time. This is based on the Unix epoch (January 1, 1970, 00:00:00 (UTC)) and measures seconds elapsed since that time. The whole of this is captured in the **time** module in Python.
https://docs.python.org/3/library/time.html?highlight=time#module-time

So, the **timestamp** we get is a float. What matters here is the difference between two consecutive events. We will use the difference between events in Lab-08.

```
def WaitingForYou():
    start = time.time()
    input("Waiting for you to hint the ENTER key ")
    return time.time() - start

>>> WaitingForYou()
>>> Waiting for you to hint the ENTER key
>>> 1.2761743068695068
```

The Joystick event is the only timed event. All the other events are logged without any time reference. That is, we also introduce a “Time” event, and this event include the current time, but nothing else.

2.5 Python modules and Python objects

In this exercise, we want to make a module name **SenseLogger.py**. All we need to do, is to create the SenseLogger.py file, include what we want in it. To use the module, you simply **import** the module. We assume here that the imported module is present in local directory[2].

The example files **JoystickLogger.py** and **HatLogger.py** does this.

Inside **SenseLogger.py** we will have a class named **Logger**. The **Logger** class will contain the actual functionality for logging the Sense HAT events. You will create **Logger** objects to do the actual logging.

[2] The module may also be part of a package or in the python system directories (we are not going into any of this).

3 The events that we shall log (WORK TO DO)

3.1 Rules for the log file

1. The log file is a text file
2. There will be only one event logged to a line
3. Events are coded as JSON serialized string objects
4. Lines with logged events shall start with "@" in the first position.
    a. This will be at position 0 in a python string. Example: line[0]
5. Lines that do no start with "@", may contain comments, etc.

3.2 SenseLogger.py

The attached **SenseLogger** module (a separate python file) defines the interface.

What you need to do is to write the missing functions and try it out on the attached **HatLogger.py** (and **JoystickLogger.py**[3]) programs.

- You write the missing **SenseLogger.py** functions.
- You test it out yourself
- Then you test it by running **HatLogger.py**

3.3 HatReplay.py

Having created a logtrail (by successfully running HatLogger.py), it is your task to complete the program **HatReplay.py**.

This program replays the event log created by **HatLogger.py**.

- You write the missing **HatReplay.py** functions.
- You verify that it does replay the log as intended

[3] This is for Lab-08