

# System Internals and Cybersecurity Miniproject

Sam Heney  
1700469

January 2021

# 1. Introduction

This project is a secure entry system that requires the user to visit a website, push a physical button which brings up a keypad on the website, then enter and submit the correct code to "unlock the door". Security is vital to this project as if an attacker could find a way to use the website to open the door remotely, or even worse without having to know the code, it would render the system useless.

## Objectives

- Implement an API for the web clients and locks to communicate
  - Ensure the API is encrypted
  - Ensure authentication is required to execute sensitive commands
- Create a centralised data storage system
  - Ensure secrets (door codes, auth keys) are stored as hashes
- Create a web interface for entering the code
  - Design for phones first, since that's the intended means of access
  - Ensure all API errors are handled gracefully
- Implement a userspace application for the Pi to communicate with the API
- Connect the two LEDs and button on a breadboard to the GPIO pins
- Implement a driver as an interface for the hardware
  - Door unlocking should be a function that could easily be changed from controlling LEDs to controlling a real lock
  - Create logical commands for the userspace application to use

## Extra Objectives

- Use linux sysadmin tools to set up the software on powerup without manual interference
- Design the system to be easily scalable (to add more locks)
- Compile the module into the kernel, rather than using insmod to load it

## 2. Procedure and Methodology

### 2.1 Cloud Layer

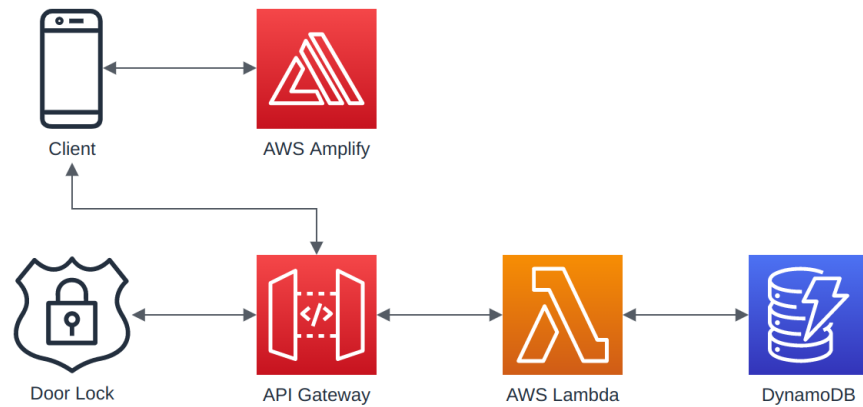


Figure 2.1: Cloud Layer Diagram

#### Websockets API with API Gateway

The first step was to implement a means of communicating between the Pi and the web clients. Websockets seemed the best choice, allowing low-latency bi-directional communication for all clients. (n.db, Amazon Web Services, Inc)

The initial routes that needed to be linked to functions are \$connect, \$disconnect and \$default. These are responsible for handling the basic behaviour of connecting and disconnecting, as well as processing any unregistered commands.

In order to be able to send messages to specific clients, a DynamoDB table was created to store all active connections. The \$connect and \$disconnect functions add and delete the connection ID to the database.

ConnectionId	ConnectionType
Y_XmkeLlrPECHBQ=	Door1WebClient
Y_Xm5elgLPEAcJQ=	Door1

Figure 2.2: Websockets connections in database

## Lambda Functions

With this in place, other lambda functions can be routed. First, a function SetType that allows clients to identify themselves. Their type is stored in the database, allowing functions to address, for example, all of the Door 1 Web Clients.

Next is SubmitEntry, which checks if the code submitted is correct. It checks whether the button has been pushed within the last 10 seconds, and if it has, it checks the code against a SHA256 hash of the correct code. If it passes all the checks, it will broadcast an unlock door message to the relevant door.

The final lambda function is ButtonPress. This function requires authentication for the button press to go through. This means that it wouldn't be possible for an attacker to remotely "press the button" by interacting with the API.

Since the websockets connection is encrypted with HTTPS, packet sniffing would not reveal the token even if the attacker got access to the network.

## React App

The final component of the Cloud layer is the Web Application used to enter the code. This application was implemented with a serverless back end, relying entirely on the API and javascript. (n.da, Amazon Web Services, Inc) The files are being hosted and served by AWS Amplify.

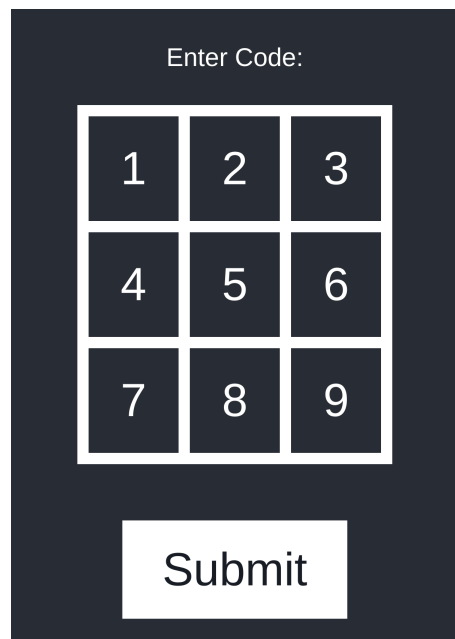


Figure 2.3: App pin entry page

When the page is loaded, it shouldn't present the keypad unless the button on the Pi has been pushed. Therefore, once the page is loaded, it only shows the keypad if it receives {"Action": "Button"} from the API.

# Press Button

Figure 2.4: App landing page

This is more of a UI control than a security control, as if an attacker watched the requests of the web app they could copy the submit request and send that to the API directly. However, there is no way to send button presses other than via the Pi, so they would still have to physically press the button within 10 seconds of submitting the code, seen in figure 2.5.

**Code must be entered  
within 10 seconds**

Figure 2.5: App timeout

Once the request is submitted to the SubmitEntry lambda via the API, there are a few different things it might return. The code could be wrong, it could be right, or it could be too late after the button was pressed. All of these cases are handled by the application and the user is informed of which occurs. Figure 2.6 shows the code submitted being rejected.

**Code Rejected**

1 2 2

Figure 2.6: Code rejected message

Finally, the application uses the URL parameter to inform the API of what door it's trying to open. Visiting the "/Door1" URL identifies the instance to the API as "Door1WebClient". This identification is also used by the SubmitEntry call, and is sent as a parameter.

For example, if "Door1" is the door ID, that is the door that is checked in the database, and the corresponding access code is retrieved. This means that this system could easily have more doors just by adding them to the database, along with their SHA256 hashed access code.

## 2.2 Userspace Layer

For the userspace application Python was used. This program needed to both act as a websockets client to the API, sending and parsing incoming messages asynchronously, while also handling communication with the kernel driver device.

### Websockets Client

Using the websockets module (n.d, Augustin, A), implementing a websockets client was fairly simple. First, a consumer/producer handler function needed to be created. This starts two threads, one for sending messages and one for receiving them.

The consumer function was the simpler of the two. In the case {"Action": "Unlock"} was received, the door would unlock. As explained in the previous section, this command could only be sent once the SubmitEntry lambda had passed all checks, making it secure. Other than that, it would log all incoming messages.

The producer function was more complex. This function needed to send a "button press" message to the API only when a signal was received from the kernel module.

First, a function was set up with a continuous loop which would await a "producer" function:

```
while True:
    # wait for producer function to finish
    message = await producer(cond)
    # function returned, button has been pressed
    log("Button Pressed")
    await websocket.send(message)
```

For the producer function to finish and return the message to send, the condition variable must be notified:

```
async def producer(cond):
    # wait for condition to be notified
    async with cond:
        await cond.wait()
    # return message code is here
```

The condition would also be passed to the signal handling function, meaning that whenever the application receives the signal from the kernel, it sends the message.

## Signal Handling

The signal handling was significantly simpler than the websockets interfacing. First, a function needed to be created that would be activated by the signal:

```
async def signal_handler(cond):
    # wait for signal to be received
    await cond.acquire()
    # when it is, notify the thread condition
    cond.notify()
    cond.release()
```

This code simply takes the condition mentioned in the previous section and notifies it, triggering the continuation of the producer function which results in the "button press" message being sent to the API. Then, during setup, the signal SIGUSR1 can be bound to the handler function:

```
# register SIGUSR1 as the one to trigger the signal handler
loop.add_signal_handler(
    signal.SIGUSR1,
    lambda: asyncio.ensure_future(signal_handler(cond))
)
```

With this, if the application receives that signal, the function will be called. For communication to the driver device, `fcntl.ioctl` calls were used to send the signals. (n.db, Python Foundation)

## systemd Service

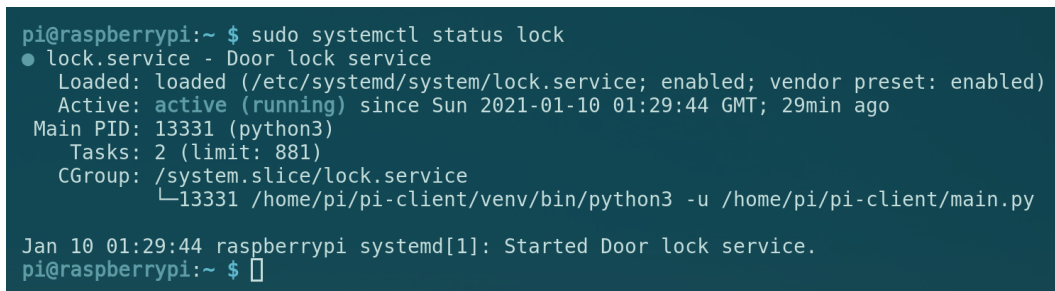
Finally, in order to start up the application on boot, a systemd unit file was written and placed in `/etc/systemd/system/lock.service`. The service runs as user and group pi:

```
[Service]
Type=simple
Restart=always
User=pi
Group=pi
```

It uses the virtual environment python interpreter with all relevant dependencies to execute the script, and redirects all output to a log file:

```
ExecStart=/home/pi/pi-client/venv/bin/python3 -u /home/pi/pi-client/main.py
StandardOutput=file:/home/pi/pi-client/lock.log
StandardError=inherit
```

The full unit file can be seen in the attached files. With this in place, the service can be activated to start on boot, and can be managed via `systemctl` as shown in figure 2.7.



```
pi@raspberrypi:~ $ sudo systemctl status lock
● lock.service - Door lock service
   Loaded: loaded (/etc/systemd/system/lock.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-01-10 01:29:44 GMT; 29min ago
     Main PID: 13331 (python3)
       Tasks: 2 (limit: 881)
      CGroup: /system.slice/lock.service
              └─13331 /home/pi/pi-client/venv/bin/python3 -u /home/pi/pi-client/main.py

Jan 10 01:29:44 raspberrypi systemd[1]: Started Door lock service.
pi@raspberrypi:~ $
```

Figure 2.7: systemd service status

## 2.3 Kernel Layer

The kernel module served two main purposes: handle button presses resulting in a signal being sent to the userspace app, and "unlocking the door" upon receiving the command to do so from the userspace app.

### On Initialisation

The first stage of initialisation is to create the `/dev/piiodev` device that will be used by the userspace application to send in commands. (2001, Salzman, P.) Additionally to the kernel code, a small udev rule was created so that the device is owned by the user running the python service, meaning root permissions aren't required to run the service.

There are two LEDs and a button to be registered via the relevant gpio commands, and both LEDs are turned on in order to show that the module has been loaded. Then, the button has to be attached to an interrupt request which calls a function. With this, each time the button is pressed, the `piio_irq_handler` function is called.

## Device Commands

Two commands were registered for the device to process. The first, `SET_PYTHON_PID`, receives and stores a PID from the python service, allowing the module to send signals to the service. It also turns off the Green LED, since if this has been called then everything is fully initialised and the door should be "locked".

The only other command is `UNLOCK_DOOR`, which when received by the userspace application starts a background thread that turns off the red LED and the green one on, waits five seconds, then switches back. This indicates the door "unlocking" for five seconds, and could easily be adapted to an electronic lock by simply replacing the code that runs the LED sequence with code that unlocks and locks the real lock.

## Button Press

The button function first goes through a debouncing check, since sometimes the button would trigger twice in rapid succession on just one press. Once passed that, it sends a signal to the python service, activating the function that will end up sending the "button pressed" message to the API. (2001, Rubini, A et al.)

## Installing and enabling the module

Finally, in order to install the module it had to be compiled with `modules_install` against the kernel headers, which would install it to the modules directory.

Once installed, the module name was added to `/etc/modules` so that it would be inserted on boot. That combined with the systemd unit for the python service ultimately results in the entire system setting itself up when powered up, with no manual interaction required.

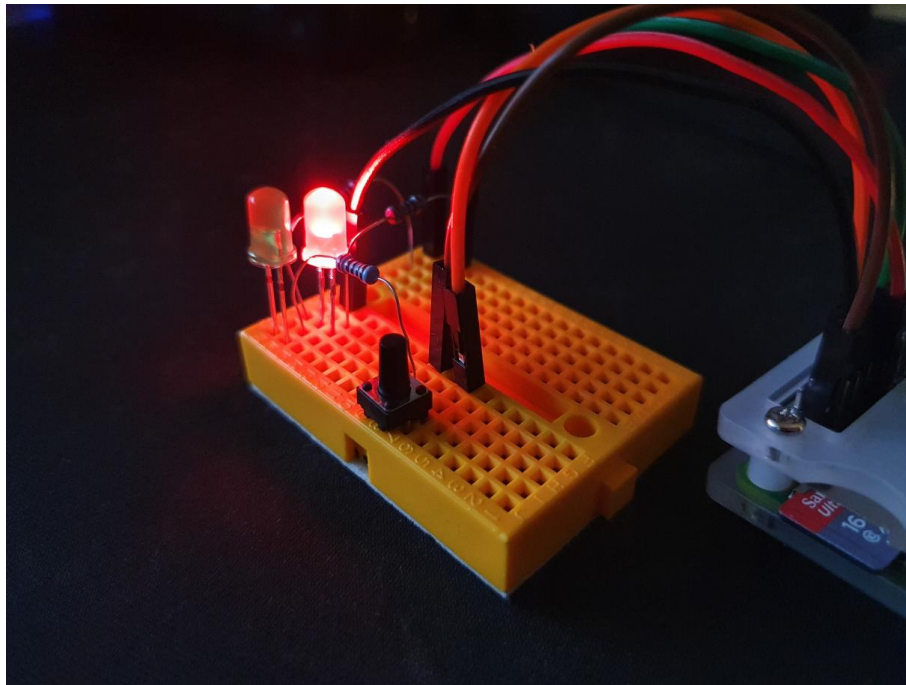


Figure 2.8: Hardware on boot



## 3. Conclusions

### **In Conclusion:**

- All of the objectives written in the introduction were fulfilled, even the optional ones.
- Throughout development there was a lot of time spent reworking the design and re-implementing components, which could have been avoided with more thorough planning.
- Good security practices were consistently maintained throughout the project.
- The AWS IoT services might have been a better choice for developing the Pi communications, but the current implementation is still scalable and fully functional.

### **Future Work**

- Acquire hardware to build more pi clients, and therefore more "doors", to test scalability.
- Get an electronic lock and implement functionality to secure a real door
- Potentially switch to using the AWS IoT services for the connection from the Pi to the API

## 4. References

Amazon Web Services, Inc. n.d. Build Your First Serverless Web App. [online] Available at: <https://aws.amazon.com/serverless/build-a-web-app/> [Accessed 11 January 2021].

Amazon Web Services, Inc. n.d. WebSocket Apis In API Gateway. [online] Available at: <https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-websocket-api-overview.html> [Accessed 11 January 2021].

Augustin, A., n.d. Websockets. [online] Available at: <https://websockets.readthedocs.io/en/stable/> [Accessed 11 January 2021].

Python Foundation. n.d. Asynchronous I/O. [online] Available at: <https://docs.python.org/3/library/asyncio.html> [Accessed 11 January 2021].

Python Foundation. n.d. The Fcntl And Ioctl System Calls. [online] Available at: <https://docs.python.org/3/library/fcntl.html> [Accessed 11 January 2021].

Salzman, P., 2001. The Linux Kernel Module Programming Guide. [online] Tldp.org. Available at: <https://tldp.org/LDP/lkmpg/2.6/html/> [Accessed 11 January 2021].

Rubini, A. and Corbet, J., 2001. Linux Device Drivers, Second Edition. [online] O'Reilly Online Learning. Available at: <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch09s03.html> [Accessed 11 January 2021].