User Manual – RCCTF

The following directions assume you have already built the car according to the manual included in the kit. The model of Raspberry Pi used should be Raspberry Pi 4.

Initial Setup

1. OS Configuration

From https://raspberrypi.com/, download the 64-bit version of "Raspberry Pi OS with desktop." Using a bootable media creator such as Balena Etcher, flash the image to a micro SD card with a capacity of at least 8GB.

Once the flash is completed, eject the card and insert it into the appropriate slot in the Raspberry Pi. Attach a monitor via one of the mini-HDMI ports, as well as a mouse and keyboard via the USB-A ports. Once the required peripherals are all connected, plug in a power source to the USB-C port on the Pi. The system should boot to the first-time setup screen. Install the system normally. You may specify your own login credentials. The default credentials are `pi:raspberry`. Don't forget to connect the Pi to a WiFi network.

Once the system has finished installing and rebooted, open a terminal window using the shortcut `CTRL+ALT+T`. Begin by updating the software with the following command:

```
sudo apt update && sudo apt -y upgrade && sudo apt -y autoremove
```

Now, your system is fully installed and up to date. In the applications menu, open "Raspberry Pi Configuration". Navigate to the "Interfaces" tab and enable SSH as well as I2C.

2. RCCTF Setup

To setup the RCCTF challenges, you will need to clone the GitHub repository into the home folder of your Pi (i.e. /home/pi/). In a terminal window, run the following command to download the necessary files:

```
git clone https://github.com/tpetersen2018/IoT-Research.git
```

Then, move the CarPlatform subfolder to the home folder:

```
mv ~/IoT-Research/CarPlatform ~/
```

Each challenge runs its own Docker image. To install Docker, run the following command:

```
sudo apt install docker.io
```

You will then need to build each docker image. Navigate to each challenge's folder within CarPlatform using the `cd` command, and run `bash docker_build.sh`. This step may take some time. After you have run all of the build scripts, use the command `docker images` to ensure that everything was built correctly. You should have a total of 8 images.

3. Network Configuration

It is very important that this is the last step. Ensure that all of the required software has been downloaded, because your Pi will no longer have access to the internet once this step is completed.

To allow other devices to communicate with the Pi, you will allow the Pi to act as an access point using a software called RaspAP. To install this software, simply run this command:

```
curl -sL https://install.raspap.com | bash
```

You will be prompted with a series of Y/N options. Proper setup is as follows:

```
ligttpd root: /var/www/html? [Y/n]: Y
Installing ligttpd directory: /var/www/html
```

```
Complete installation with these values? [Y/n]: Y
...
Enable HttpOnly for session cookies (Recommended)? [Y/n]: Y
...
Enable RaspAP control service (Recommended)? [Y/n]: Y
...
Install ad blocking and enable list management? [Y/n]: N
Install OpenVPN and enable client configuration? [Y/n]: N
Install WireGuard and enable VPN tunnel configuration? [Y/n]: N
...
The system needs to be rebooted as a final step. Reboot now? [Y/n]: Y
```

After the system has rebooted, you will need to configure the RaspAP settings. On the Pi, open a web browser and simply type `localhost` into the address bar. The default login credentials are `admin:secret`. On the sidebar, navigate to the Hotspot menu. Go to the tab labeled "Basic". Ensure the selected Interface is `wlan0`. Here, you may change the SSID (network name). The recommended naming convention is "RCCTF-<Color><Third byte of IP>" for example: `RCCTF-Red146`. It is also recommended that you configure the hotspot to run on channel 11, but it is not strictly necessary. Click on the "Security" tab. If you would like to require a password to connect to the hotspot, you may do this here. Otherwise, change "Security type" to "None".

On the sidebar, click the option labeled "DHCP Server". In the "Server settings" tab, ensure the "Interface" option is set to wlan0. Select "Static IP" rather than "DHCP". Under "Static IP options", you may specify a custom IP address to use for your Pi. The default value is `10.3.141.1`. For the subnet mask, use `255.255.255.0`. The default gateway will be the previously specified IP address. Enable the option labeled "Enable DHCP for this interface." Set the starting and IP addresses to <First three bytes of IP>.50 and <First three bytes of IP>.255 respectively (e.g. `10.3.146.50` and `10.3.146.255`).

On the sidebar, navigate to the "Authentication" menu. You may change the login credentials for the RaspAP config here if you so choose.

Finally, go to the "System" menu on the sidebar, then click on the "Advanced" tab. Change the web server port. It is recommended to use port `8080`, but most other ports will be fine as well. Do not use port `22` or `80`. These are reserved for SSH and RCCTF's "Web" challenge.

Save the settings and reboot the Pi. You are now ready to start solving challenges!

Starting a challenge

The entire car is powered by the Raspberry Pi. Simply connect the Pi to a power source (i.e. the portable charger/battery) via the USB-C port on the board to power it on. Please allow at least 30 seconds for the system to boot properly.

Once fully booted, the Pi will serve as a wireless access point. Using your own computer, connect to the WiFi network corresponding to the car you are using. For example: `RCCTF-Red146`

Your computer and the car are now able to communicate with each other over the network.

To begin solving challenges, first connect to the Pi via SSH. Enter the following command in the terminal of your computer:

```
ssh <username>@<IP address>
```

Once you have successfully logged in, change directories into the CarPlatform folder using cd. From here, you can change directories into the folder that corresponds to the challenge you wish to play. Simply use the following command to begin the challenge:

```
bash docker_run.sh
```

See “Challenge Solutions” for instrcutions on how to conquer each individual challenge.

Challenge Solutions

- **UDP Replay**

  There is a UDP server running on port 31337. It will wait for a connection, and listen for commands to send to the motors.

  ```
  nc –u <IP address> 31337
  ```

  You're connected! Simply send commands such as "u" or "d" to the server to drive the car.

- **Cocoa 1**

  Once again, there is a UDP server running on port 31337. By examining the source code, we can see that there is some type of authentication happening. In order to gain control of the car, the first message sent to the server from our client must be "admin".

  ```
  nc –u <IP address> 31337
  ```

  Once a connection has been established, simply type "admin". The server will respond with "Authentication successful". Now you can simply control the car with the arrow keys on your keyboard.

- **Cryptography**

  This challenge is once again running a UDP server, but this time it's on port 33047. Connect to it in the same way you did in the previous challenges.

  ```
  nc –u <IP address> 33047
  ```

  The concept behind this challenge is fairly simple. Any message you send to the server will be converted to a string of bytes. These bytes are then passed through a function which will encrypt the message using the XOR operation. The key to this operation is the hex byte `\xB7`. The server will check the encrypted message to see if it is a valid command (u, d, l, r, t).

CyberChef is a very useful tool when it comes to cryptography challenges. Simply add the XOR function to the recipe with the key B7 (HEX), and type the message you want to encrypt in the "Input" box. The output will be the encrypted message you can send to the server.

The resulting encrypted message may have some non-standard characters that cannot be typed normall. In this case, add the "To Hex" function to your recipe in CyberChef, and copy the hex byte that is produced. To send your encrypted message to the server, use one of the following commands:

```
echo -e "\x<encrypted byte>" | nc -u <IP> 33047
python -c 'print("\x<encrypted byte>")' | nc -u <IP> 33047
```

- **Reverse Engineering**

This challenge has a TCP server listening on port 1337. Using either telnet or netcat, connect like so:

```
nc <IP> 1337
```

You are given a prompt that reads "Please authenticate with your valid driver license >>>". Submitting an incorrect license causes the program to close the connection, and you'll have to reconnect.

The basis of this challenge is a program simply called "car". Depending on how you have chosen to run the challenge, you may either examine the source code directly or disassemble the compiled binary with a program like BinaryNinja or Ghidra. Either way, you will gain insight into how the license is verified.

A good place to start in any reverse engineering challenge is by analyzing the function `main()`. It can clearly be seen that the only thing this function does is call another function called `authenticate()`. Looking at this function, we can see that this is where the prompt is displayed. More importantly, this function seems to return a different response depending on the result of another function, called `license_check()`, which is only called if the license contains exactly 8 characters.

This function can be divided into three parts. The first part is a simple `if` statement which checks to make sure that the first two characters of the license are "FL". Next, there is a for loop which iterates through the rest of the remaining characters. On each iteration, the bits representing the value of the res variable are shifted to the left four times, then the value of the current character in the license is added to that variable. After the for loop is completed, there is one more `if-else` statement that will return true if and only if the value of `res` is `54946352`.

Now that we understand how the program works, we need to find a way to perform these operations in reverse and find a valid license.

One method of solving this is with a symbolic execution engine such as Z3 or angr. Writing a simple python script which creates a Z3 solver with all of the necessary constraints easily gives us the valid license: `FL133700`

- **Buffer Overflow**

  Using telnet or netcat, connect to the TCP server on port 1337.

  ```
  nc <IP> 1337
  ```

  You are given a message that reads "We are sorry, but our car service is not available at this time. Please provide your contact info and we'll get back to you as soon as possible >>>"

  Analyze the program either by using a program like BinaryNinja or Ghidra, or by simply reading the source code. The main function calls another function called `authenticate()`. This function will only return true if the value of the access variable is "GRANTED". Unfortunately, there doesn't appear to be a way to change that value within the program.

  Thankfully, as hackers, we don't need to follow the rules. There is a very simple way to break this program. If you look closely at this function, you can see that the `contact` variable has been allocated 16 bytes of data, but the call to `scanf()` that reads our input will accept 24 bytes of data. This means that the program is vulnerable to a buffer overflow attack. If we get the program to send more than 16 bytes of data to `contact`, the remaining bytes will overwrite whatever is on the next level of the stack in memory. We can see that `access` is declared immediately after `contact`, meaning `access` would come immediately after it on the stack. Therefore, any excess data sent to `contact` will begin to `overwrite` the value of access.

We know that we need at least 16 bytes to reach the variable we want to change, so we will pad our input with 16 random characters. After the padding, we add the word GRANTED. Now that we have all of the info we need, our final payload should look something like this: `AAAAAAAAAAAAAAAAGRANTED`

Submit the payload, and you've got control of the car!

- **Buffer Overflow 2**

Just like the previous challenge, there is a TCP server running on port 1337. Connect using telnet or netcat, and you will see the following message:

```
We are sorry, but our car service is not available at this time.
Please provide your contact info and we'll get back to you as
soon as possible >>>
```

Giving any kind of valid input will close the connection. Begin by examining the program in a disassembler. After a careful inspection of the machine instructions, you can see that, like before, `main()` calls a function called `authenticate()`. Once again, a call to `fgets()` allows 32 bytes of input to be sent to a variable `contact` which only has a capacity for 16 bytes. This means we can overflow the buffer.

But `contact` is the only variable declared in `authenticate()`. How can an overflow help us here? Thankfully, the stack holds more than just variables. With the right amount of padding, we can overwrite the return address of the function. Looking back at our disassembly, we can see that the function `driver_menu()` still exists in the program. Take note of the memory address for this function. If we overflow the buffer in `authenticate()`, we can overwrite the return address with the address of `driver_menu()` and take control of the car anyway.

In memory, the return address of `authenticate()` is 24 bytes after `contact`. This means we will pad our input with 24 random characters, followed by the address of `driver_menu()`. The final payload will not be human-readable, as the target address will contain bytes which do not correspond to any ASCII characters. To generate this payload, use the following python command:

```
python –c ‘print(“A”*24 + “\x70\x0d\x40”)’ > payload
```

Notice how the bytes in the address are in reverse order. This is because the Raspberry Pi is running on a little-endian architecture.

To send the payload to the server, use the cat command and pipe the output into your usual netcat connection:

```
cat payload – | nc <IP> 1337
```

You should now have access to the driver menu, and you can send commands to the car using the arrow keys.

- **WebApp Exploitation**

  In a web browser, type the IP address of the car into the address bar. You will be greeted with a website containing four arrow buttons which presumably control the car. Unfortunately, clicking the buttons returns the following error message: “You’re not authorized to drive this vehicle…”

  Using “Inspect Element” or by simply navigating to your browser’s developer tools, you can view the web page’s HTML source code. There are two sections of particular interest. The page is running some Javascript code, as indicated by the <script> tags. The first script seems to store a certain value in your browser’s Local Storage. The second would appear to check that value and use it to decide whether you should be allowed to drive the car.

  Using your browser’s developer tools, access the local storage. There is a key “authorized_to_drive” with the value set to “False”. Change this value to “True”. Now, on the webpage, try using the arrow buttons again.

  Success!