Embedded Security (Summer Term 2019)
Dr. Stefan Nürnberger
CISPA Helmholtz Center for Information Security / Saarland University

Solution of Exercise Sheet #2

1 General info (0 points)

For this exercise **you will work with shared hardware! Start working early!** The hardware is located in the student room 2.17 at CISPA. There you will find an instrument cluster from a *Renault Zoe* (task 2) and from a *Seat Ibiza* (task 3). For the third task you can communicate to the Arduino via the Ethernet HUB we provide – details on how to send/receive data in the tasks. Please keep in mind that if multiple teams send data at the same time, collisions might occur: **synchronise to avoid such scenarios**. Please, **do not overwrite the firmware on the Arduinos!**

Finally, the CISPA building is accessible only from 9am to 6pm (you can stay inside until later – but you won't be able to enter again if you go out) during weekdays. If you want to have access outside these hours and during weekends, send an email to **sohyeon.park@cispa.saarland** with your **UdS card number** which is located on the top-left corner of the card (above the picture).

2 Teach me how to drive (6 points)

Since the day you helped out your friend to extract the key from the compromised device, your name is on everyone's lips. You recently upgraded your office and are now relaxing in a hammock, as the telephone wakes you from your dreams. The news of your skills got around quickly, so the car manufacturer from the next city, *FolksCar*, asks you for your help. They want to upgrade their car fleet to drive autonomously and have already developed a perfect self-driving software. Unfortunately they lost the documentation of their CAN bus protocol (rumor has it that there was never one), so they are unable to send the command which displays the correct speed and state of car to the driver. You agree kindly to work for them, under the condition that they send you a dump such that you can work on the task at your office.

For this task, we provide you a file containing a snapshot of a Renault Zoe CAN traffic. Each row in the file encodes a can frame: the first 3 characters are the frame ID, followed by a #, followed by up to 16 HEX characters which build 8 bytes pairwise. The related Arduino receives a UDP packet with a payload that represents a CAN frame. The formatting is the same as the file (example: 2b9#deadbeefc0cac01a). You can send UDP packets to the Arduino at IP **192.168.1.176** and port **8888** (just connect with the provided ethernet cable).

If something goes wrong, the Arduino responds with a debug UDP packet with the error message back to your IP at port 5000. You need a static ip for this exercise. If you are unfamiliar with it, you may want to look up how to configure a static ip.

(2 points) (a) As a first step, you need to make the cluster work. If you do everything right the cluster should turn (and stay) ON (some errors might still be visible – ignore them). List the frequencies you choose for each ID and motivate your choice and reasoning. Reasonable frequencies are between 10 to 500 milliseconds.

Solution
As CAN is a broadcast network, the correct solution would be to broadcast all frames in a cyclic way – this would simulate the complete car. The frequencies can be assigned based on the frame priority (e.g, 10ms for high priority IDs, 500ms for low priority IDs). Luckily the ZOE cluster turns on, albeit with errors, at the presence of a specific frame – 0x35C (the frequency can be anything below 1 second).

(4 points) (b) You need to discover which ID the speedometer listens to and the frame that sets the speed to 136km/h. Similarly, you need to discover which bit of frame 0x652 makes the writing "ready" disappear from the cluster. Briefly and precisely explain how you got there and how the data is interpreted.

Solution

To find the frame containing the speed, one could simply stop sending each frame for few seconds and see how the cluster reacts. Upon identifying the frame (0x5D7), the bits can be found by fuzzing each byte individually: hence you find that the first 12 bits set the speed. Anything from 0x332 to 0x338 would set the speed to 136 km/h.

You can apply the same to the 0x652 frame to find the bit(s) that remove the "ready" label. You will find that setting the 10^{th} bit removes the "ready".

3 Baby you can drive my car! (12 points)

You successfully completed *FolksCar*'s job and as a reward they gifted you their newest car, the model *Croquet 7*. The self-driving works like a charm; the only drawback is that you can no longer drive on your own. You know, however, that the car would let you drive in case of malfunction. Luckily, the car has a built-in WiFi function to allow a smartphone enabling the cars seat heating function remotely. You now search for a "hidden function" to fake the engine's temperature such that it thinks it's malfunctioning.

You got your hands on some code driving the media system. It is included in the zip file. **The task involves buffer overflows: please reset (by pushing the reset button) the Arduino if it stops working.**

(1 point) (a) Briefly explain what buffer overflows are.

Solution

A buffer overflow is an attack where you write into a part of memory that you're not supposed to write to by writing into a previous part of memory then exceeding its length. This usually happens when there are no length checks on writes to check if the write will exceed the buffer size or not.

An example would be when writing into a buffer using a loop whose length depends on the data to write rather than the length of the destination buffer. If the length of the string is longer than the length of the buffer, then a buffer overflow would occur.

```
for (int i = 0; i < len(str); i++)
    buf[i] = str[i];
```

(1 point) (b) Briefly explain how one can find vulnerabilities that can be exploited using a buffer overflow.

Solution

If the source code is available: check for buffer access with dynamic indices or static indices that are larger than the size. Otherwise, try long inputs and check if the program crashes or otherwise behaves not normal.

(1 point) (c) Briefly explain two ways to prevent buffer overflows.

Solution
Whenever writing to a buffer, it is best practise to *always* check the size of the buffer, and to make sure to *never* write above that. Another way is to use only safe versions of functions such as "snprintf" vs "sprintf" in C/C++ to print into a string. Other ways include using operating system layer features that check that pointers or the stack are not overwritten when a function returns, or stack canaries.

(2 points) (d) The Arduino is broadcasting each frame it transmits on IP address 255.255.255.255 and port 5000 (the frame format is the same as before). Your first task is to parse these packets and make a table listing (i) all transmitted IDs, (ii) whether the payload changes or not, and (iii) the (approximate) transmission frequency.

Solution
You can use Wireshark, or python, to collect packets for for 30-100 seconds. The IDs, the payloads statistics and frequencies can be easily extracted with a basic python script.

(1 point) (e) By just analysing the logs, which frame(s) might be transmitting the speed? Motivate your reasoning.

Solution
Since the speed in the cluster changes one can discard all static frames. The remaining can be narrowed by checking how many bits change in average, the frame priority, and the frame frequency. In fact, the speed displayed to the driver has low priority but relatively high frequency. In this case, the speed frame is 0x5A0. Any reasonable deduction suffice.

(1 point) (f) Again, by just analysing the logs, which frame(s) are most likely to contain the airbag status (ON/OFF)? Motivate your reasoning.

Solution
The airbag is critical for safety. It is very likely that the frame containing the airbag state has high priority and high frequency. The highest priority frame is 0x050, which also has very high frequency. Any reasonable deduction suffice.

(5 points) (g) Now analyse the code which is part of the infotainment system.

- State at which line the program is vulnerable to buffer overflows.
- Sketch your attack and what your benefit would be.
- Finally, state the payload of the UDP packet which lets you send the payload **0x33, 0xed, 0x00, 0x00, 0x00, 0x58, 0x52, 0x00** with the id **0x288** as a CAN message.
- If sent in the right frequency for several times, the engine temperature should overheat.

Figure 1: Control light for engine temperature

We highly recommend you to actually mount the attack, as a not working approach results in no points. If your attack was successful, the engine overheat light should come on. **The Arduino can be contacted at IP 192.168.1.177 and port 8888 – CAN packets and error messages are encoded the same as before**. You can connect to the Arduino via the switch.

Solution
You need to write a message that will overflow into the "allowed" array: this can occur because the function "parseIntoCanBuf" will write into the "canBuf" buffer without checking whether we're exceeding its length or not. Any input that exceed the "canBuf" length will overflow into the "allowed" array: this allows us to replace any entry in the list. There are two things to note: i) the size of *int* in Arduino is 2 bytes (in x32-x64 architectures is generally 4 bytes), and ii) the encoding is little-endian.

There are many possible messages you can send to case the overflow. One example is: "000#00000000000000000000000008802"

After causing the overflow, the ID 0x288 will be present in the "allowed" array – this will allow to bypass the ID security check. At this point, sending the overheating message frequently enough (50ms) will cause the overheat light to turn on.

4 The bus CAN do. (8 points)

(1 point) (a) Briefly state and explain a reason why messages must be transmitted periodically, even though the state of the corresponding device did not change.

Solution
Cyclic messages allows to detect faults very effectively. For example, the airbag should deploy as soon as possible in case of accident, even if the CAN bus fails. As such, ECUs interpret the absence of messages as "critical faults" (e.g., cut cable) and act accordingly (e.g., the airbag fires).

(1 point) (b) When designing the CAN architecture, the manufacturer has to ensure that bus utilisation is never too high when designing all messages. Given the frames and frequencies you utilised for exercise **2**: what is the CAN bus utilisation (in %) on a 500kbit/s bus (you may ignore bit stuffing).

Solution
This will highly depend on the values you chose in exercise 2, but the intuition is the same. A bus operating at a frequency of 500kbit/s transfers 500,000 bits per second – this implies a 2 microseconds bit time. Assuming a header size of 47 bits (standard header) and a calculated bit time of 2 microseconds[a], the bus utilisation can be then computed in various ways. The most straight-forward is to sum each frame's bit-length (header+payload) times its frequency; this number can be used to divide the maximum bitrate. In case of extended header size of 64 bits the reasoning is the same. Finally, one can assume all frames to be extended and 8 bytes long – this would provide an upper bound.

[a] `https://en.wikipedia.org/wiki/CAN_bus#Frames`

(1 point) (c) In task **2** we learned how to manipulate CAN frames in order to show spoofed information to the driver. In most cars, our attack would not set the actual speed of a car, i.e. it would not accelerate or brake according to your spoofed speed. Briefly explain why.

Solution
The frame ID is only responsible for the value of the speedometer, not the actual speed – changing it would't affect the car behaviour. Moreover, most cars have multiple CAN busses: for powertrain, for infotainment system, etc. These busses are interconnected via a gateway which is like a firewall.

(1 point) (d) *FolksCar* were impressed by your ability to spoof the speed of the car. However, they are now afraid that if someone finds a flaw in their quality software^{TM}, someone might be able to set the speed of the car while someone is driving. They instructed you again to find solutions for this problem. You learned that the CAN bus protocol is old and it is not possible to verify the integrity of the message. As cars are getting more and more connected, this topic becomes more and more interesting, hence researchers proposed different ways of tackling this problem. One such technique is explained here[1].
Briefly explain what is used to secure the integrity of the messages.

Solution
The authors used an HMAC of the payload of the packet, sent in a separate, lower-priority, frame. The receiver can itself compute the HMAC and verify if the original packet has been tampered with. The solution is backwards compatible and doesn't require the manufacturer to change the protocol. Furthermore, given that the HMAC is sent in a lower-priority frame, the solution doesn't affect CAN safety.

(4 points) (e) The authors decided to use symmetric cryptography with static keys. List one advantage and one disadvantage each for instead using asymmetric cryptography or using a key exchange to establish a common key.

Solution
Asymmetric cryptography:
CONS: Is slower than symmetric cryptography, and requires more complex set-ups – useful only for key exchanges.
PROS: Private keys don't need to be shared, so a compromised device will not leak all the keys for the other devices.
Key exchange:
CONS: The key exchange needs to take place before the car can operate and a failing device needs to redo a key exchange, which leads to (short) unavailability periods.
PROS: The keys are no longer static, hence leaking a key does not give you the possibility to create signed messages forever, but only until the car reboots. Further, a device can be substituted without having to replace the corresponding key on all other devices.

[1] `https://link.springer.com/chapter/10.1007/978-3-662-53140-2_6`, only free in eduroam