BUFFER OVERFLOWS

WHEN DO OVERFLOWS OCCUR?

- When values are written outside of the specified bounds
- Ex:
  - char myStr[100]; // starts at 0x00, ends at 0x64
  - if we read myStr[16], we are ok: 0x16 > 0x00 && 0x16 < 0x64
  - If we read myStr[120], we MAY be ok: 0x78 is out of bounds
  - If we write myStr[120], we are likely to crash if more variable are not declared
- We cannot write at an index less than 0 or more than 99. **100 is out of bounds**
- We CAN write at any index in-between, inclusive of 0 and 99

HOW TO PREVENT OVERFLOWS

- Know the size of the array we are working with
- Do not hard-code address values unless we are working with hardware
- Never assume something succeeded unless you checked it
  - Ex
    - You allocated 100 bytes for an array with malloc (it succeeded), but then you reallocated for a size of 2000 bytes.
    - You did not check after the 2000-byte array and you write to position 100; since the allocation failed, you are not writing outside of bounds. Arguably, this falls under CWE-252: Unchecked Return Value, but we continue to write outside. The underlying cause is CWE-252.
- Moral of the story: stay in bounds and validate return values.

USING MACROS

- If you are going to work with an array of size 100, use a macro constant to expand to 100
- That way, we can:

```
int arr[BUF_SIZE];
for (int i = 0; i < BUF_SIZE; ++i)
      ;
```

- With the above code, we can change BUF_SIZE without possibly reading/writing too much or possibly short depending on the change
- Macros can make code a lot simpler, less bug prone, and easier to mange
  - *When used incorrectly, it can make code worse and introduce bugs!*

EXPLAINING CODE IN GH/SAMPLE CODE/REVIEW/BUFFER OVERFLOW

- BuffOverflow.c/.h contain the code pertaining to a possible overflow
- The for loop on line 53 shows how we can easily write past the buffer's length when we use incorrect bounds. This should be straight-forward and easy to see how using the length of the input (str1CopyValue) into a size smaller (str1) can override data in neighboring bits.
- After that, we get to a very specific example where we target changing a variable after looking at all the variable's addresses. I modify the value of variable 'a' by using str1, str2, and str3. I did this by taking the difference of the addresses. From there, I plug in the difference as the index and change the value of a (see printed values in the log files).
- Note how we can use negative index values (don't do this in practice, I am making a point that buffer overflows or underflows can override data)

(CONT.) EXPLAINING CODE IN GH/SAMPLE CODE/REVIEW/BUFFER OVERFLOW

- Heart Bleed
- This is a method that attackers use to read more bytes than what they sent on an echo server. This could leak passwords, usernames, and other sensitive info. It is like verbally saying hello three times, but telling the other person you said it five times, so you get “hello hello hello SENSITIVE INFO” as the response.
- I simulate this by reading from 0x100000000 to the end of str1CopyValue which is in read-only memory. My computer likely uses different memory addresses than your devices.
  - If I read one byte before 0x100000000 or after str1CopyValue, the program crashed
- As such, HEART_BLEED_MAX is based on my system, and you may be able to read more than I can.

EXPLORATIONS FOR CYBER-SEC

- If Heart Bleed is interesting to you, look at the address of str1CopyValue then see what values you can read around that.
  - Is str1CopyValue the start, end, or somewhere in-between of your readable range?
  - Can you read earlier than 0x100000000?
- If the variables are lined up in your memory, is there a loop you can write to modify all variables?
- If you have 10 int variables, how can you modify others only using the first int?