Ben Brandhorst

SDEV 325 6380

Detecting Software Vulnerabilities

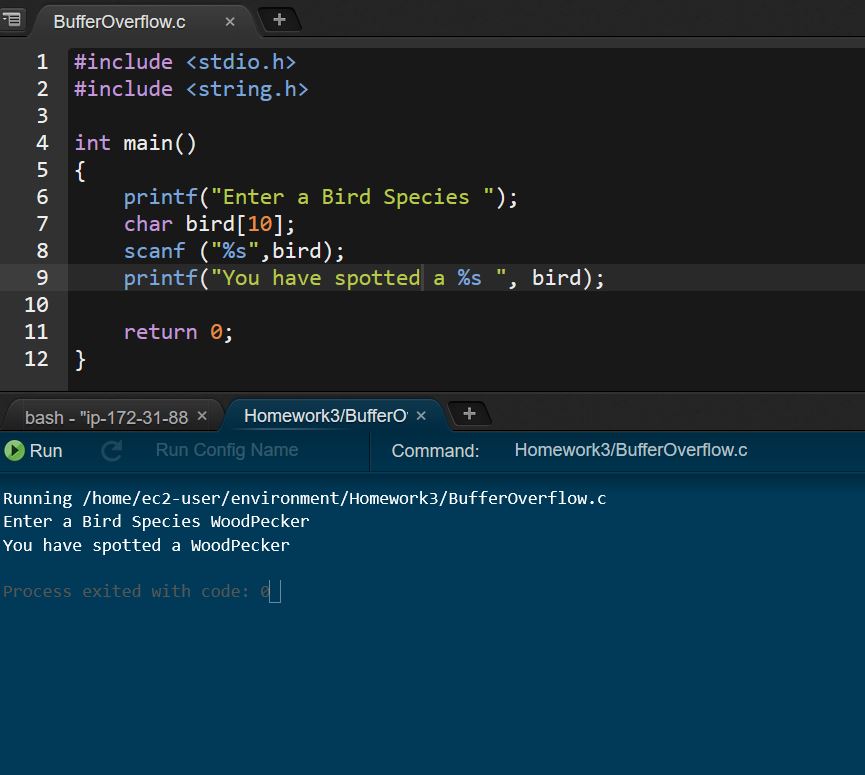
Professor Luke Donoho

09/14/2019

**Demonstrating Risky Resource Management**

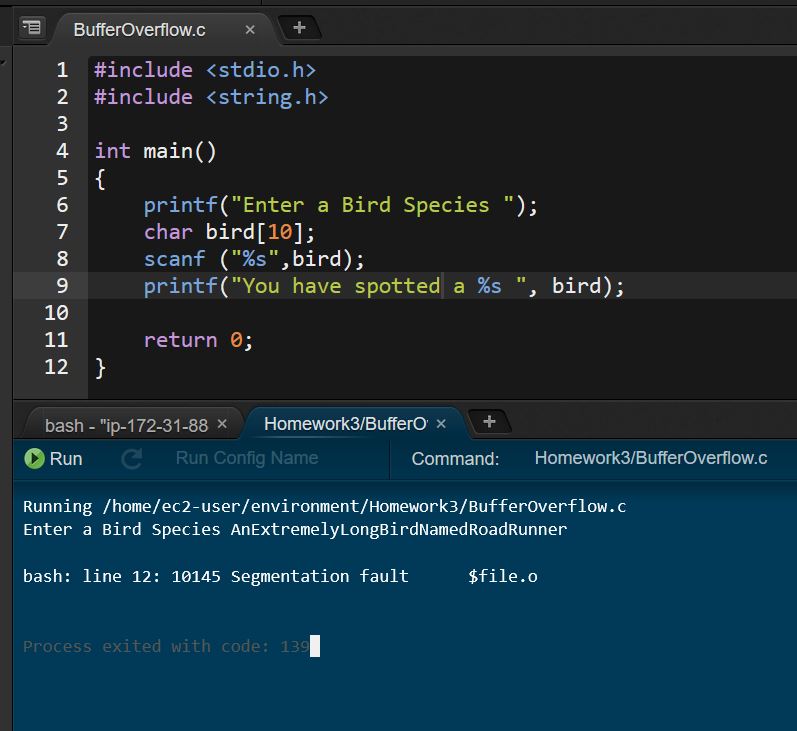
**Example #1**

**CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**

****

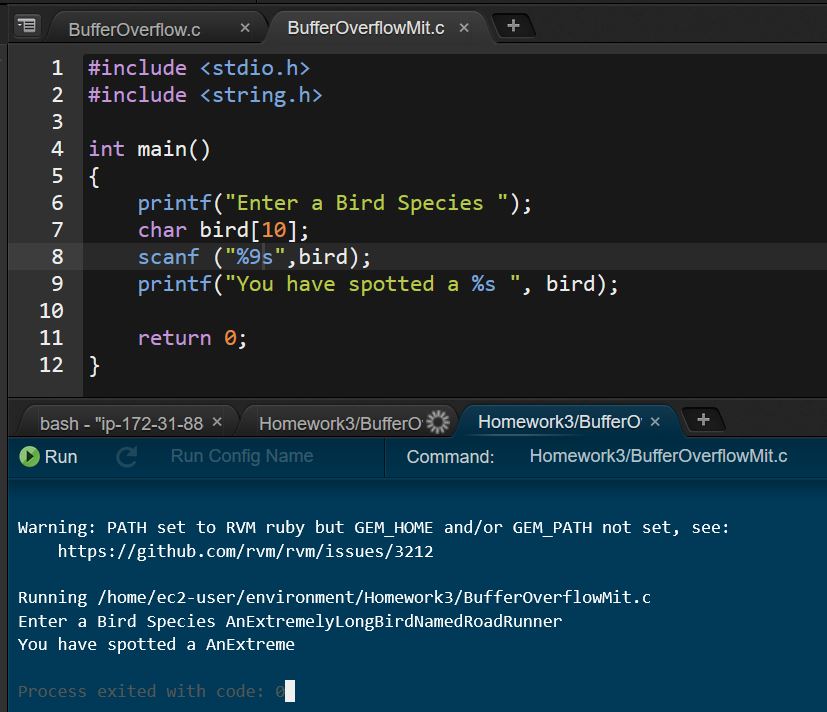
The code in this example is vulnerable to the classic buffer overflow exploitation. This is because the temporary storage allocated for bird species names has 10 bytes of buffer reserved, while the user input is not restricted in size. This means that it is possible for a user to provide an input larger than the space reserved for it. The user input is then copied to the buffer without any checks. If the input exceeds the allocated buffer size, results can span from program crashes, files damaged, or the unauthorized access of private information.

**Example of Buffer Overflow Crashing a Simple Program**

****

In the example above the input bird species name exceeded the buffer size and caused the program to crash with a segmentation fault error. This happened because the program attempted to access a memory location that it is not allowed to access and crashed when it was denied access to that location. It is possible for more advanced attackers to use the “overflow” to inject or execute malicious code.

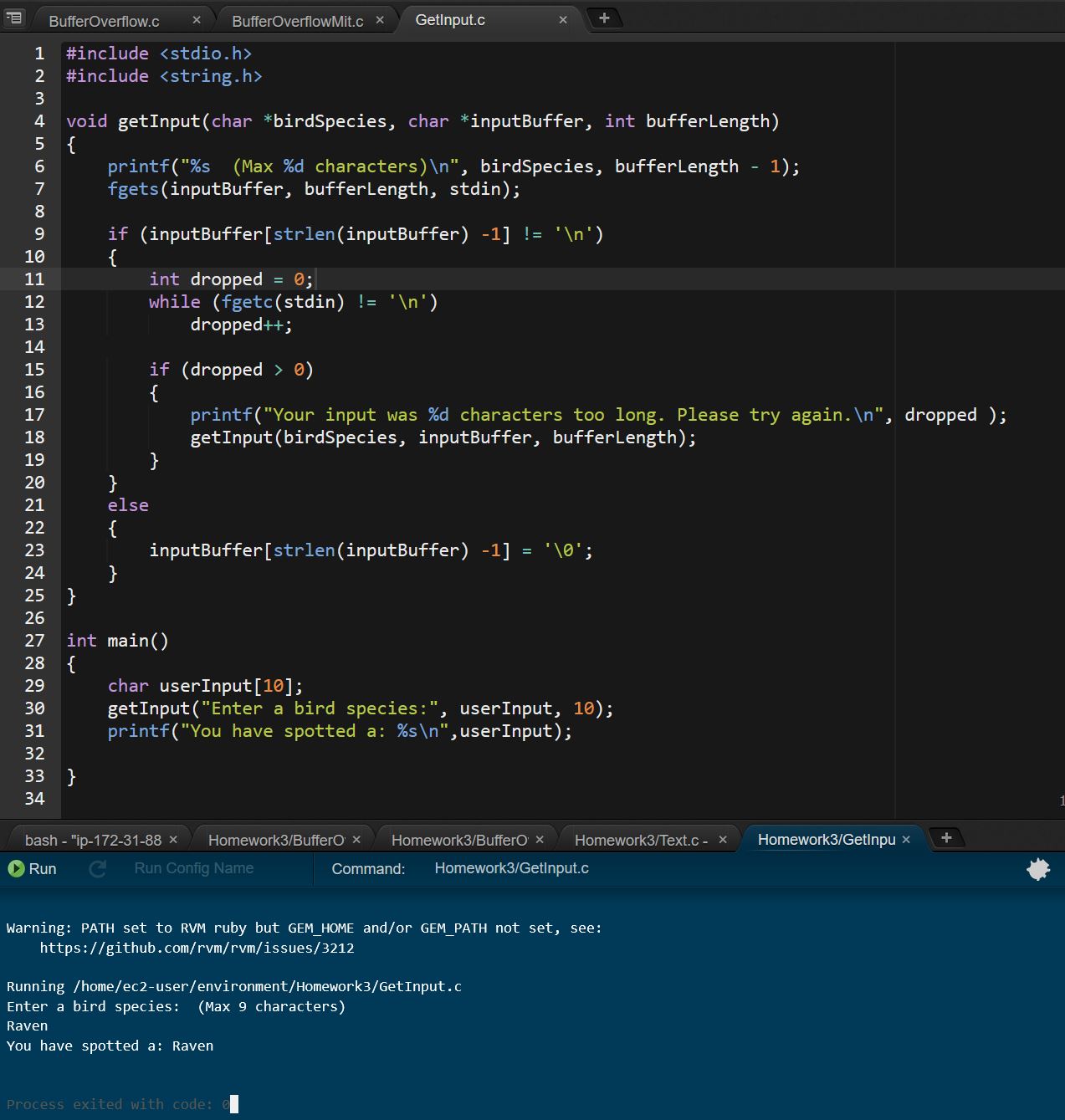
**Example of Buffer Overflow Mitigation Through Truncated User Input**



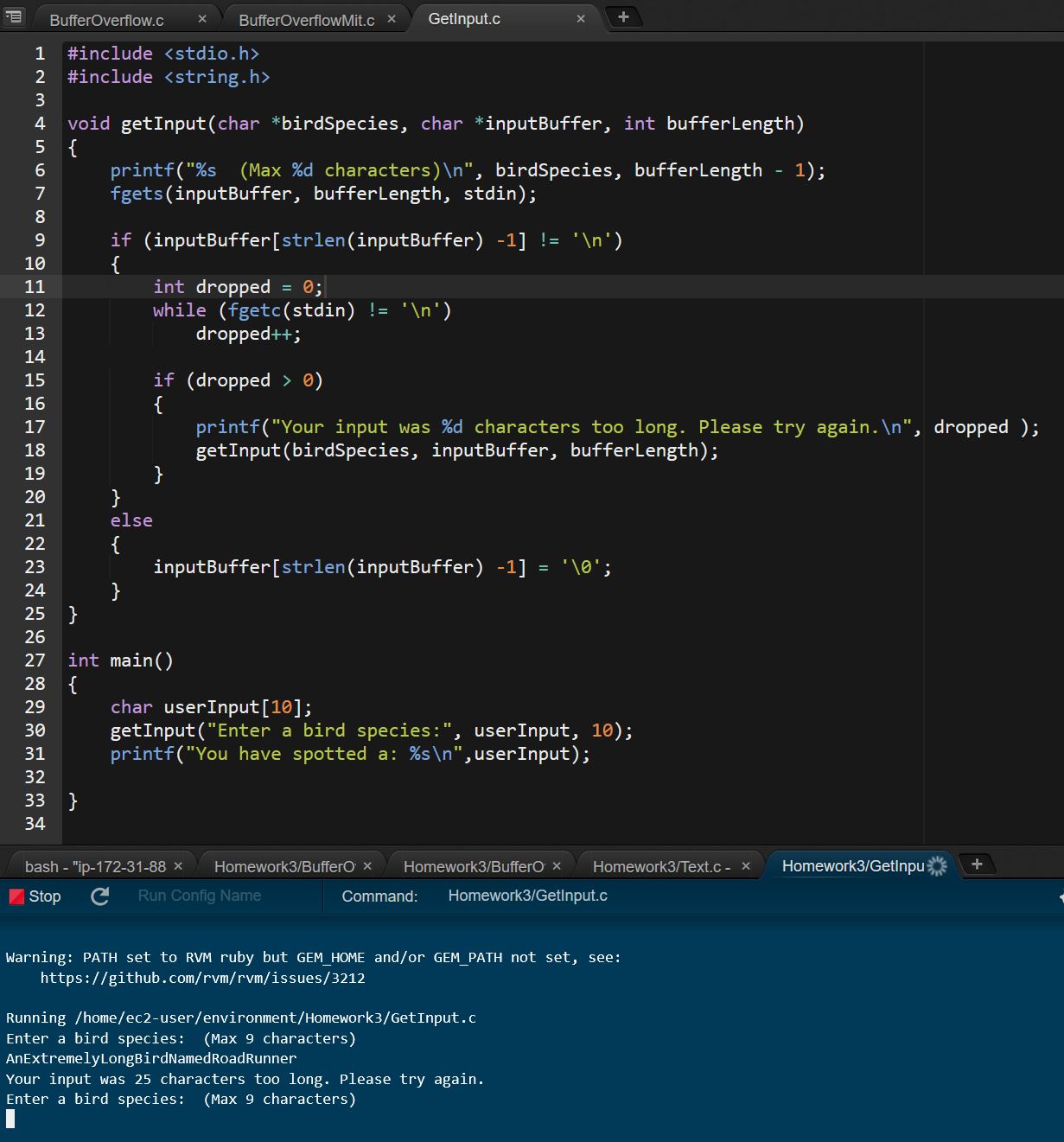
In the example above we’ve modified the scanf() function to truncate user input longer than 9 bytes. In order for this to work properly, we must account for the 0-terminator. This means that to prevent the possibility of a buffer overflow the scanf() function is set to read 1 less byte than the size of the buffer. In this case the buffer is **bird[10]** which means scanf() is modified like: **scanf (“%9,” bird); .** Now any excess characters are simply not scanned and the input is guaranteed to fit within the buffer. This mitigation only protects against buffer overflow and does not address other vulnerabilities.

However, silently disregarding the disallowed characters seems like bad program design. I think a more reasonable approach would be to tell users when their input is too long and provide a chance to resubmit. I only have a baseline level of familiarity with C, so I used the framework of an answer I found on [StackOverflow](https://stackoverflow.com/questions/7880141/how-do-i-check-length-of-user-input-in-c) for the following example:

**Example Using getInput() to Specify Maximum Input Size**



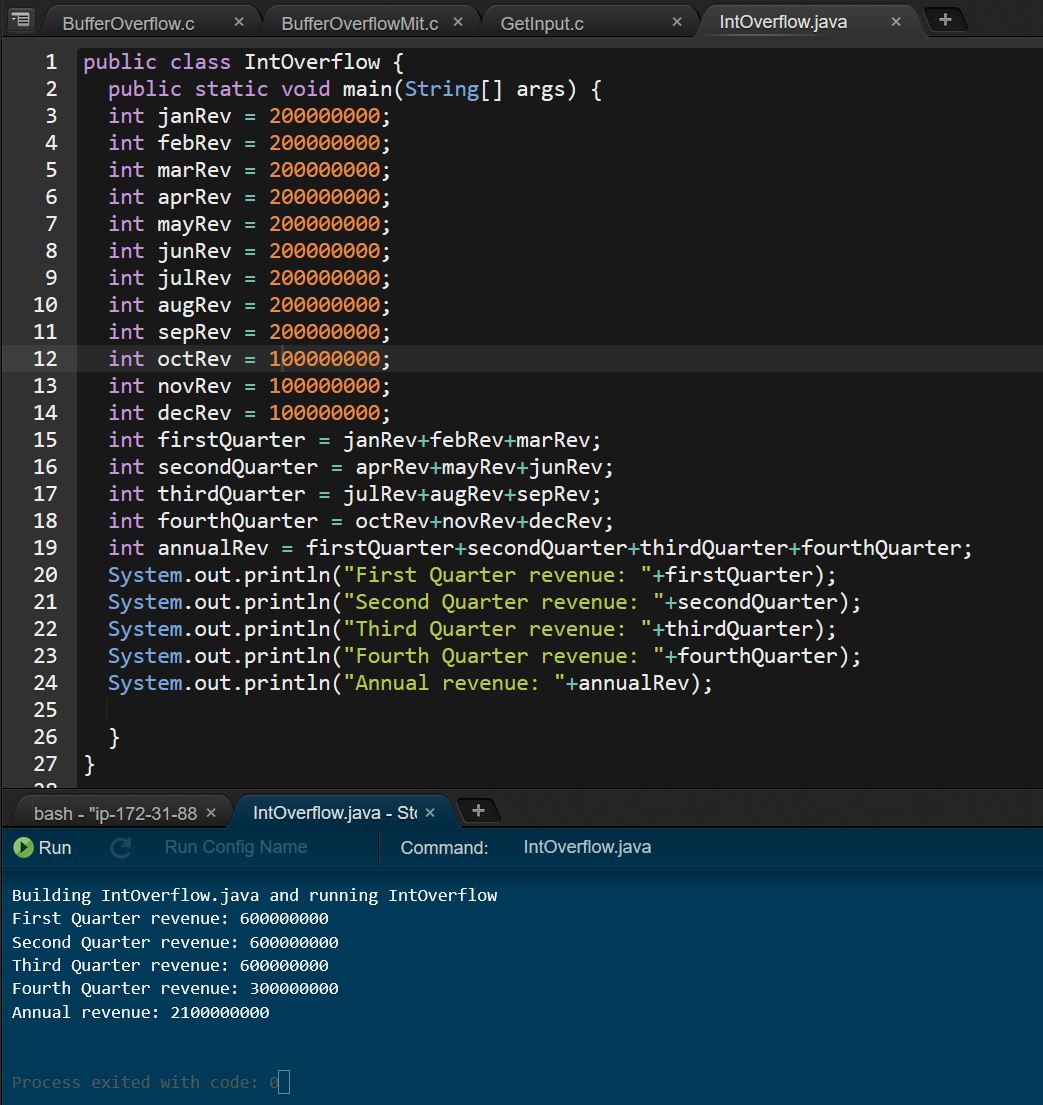
**Example When User Input Is Larger Than Allowed**

****

As you can see the use of **getInput()** instead of **scanF()** allows us to determine the length of the input before it is copied to the buffer. This allows us to write code that will check the size of the input and inform users their input was not within the allowed parameters instead of just ignoring a portion of the input.

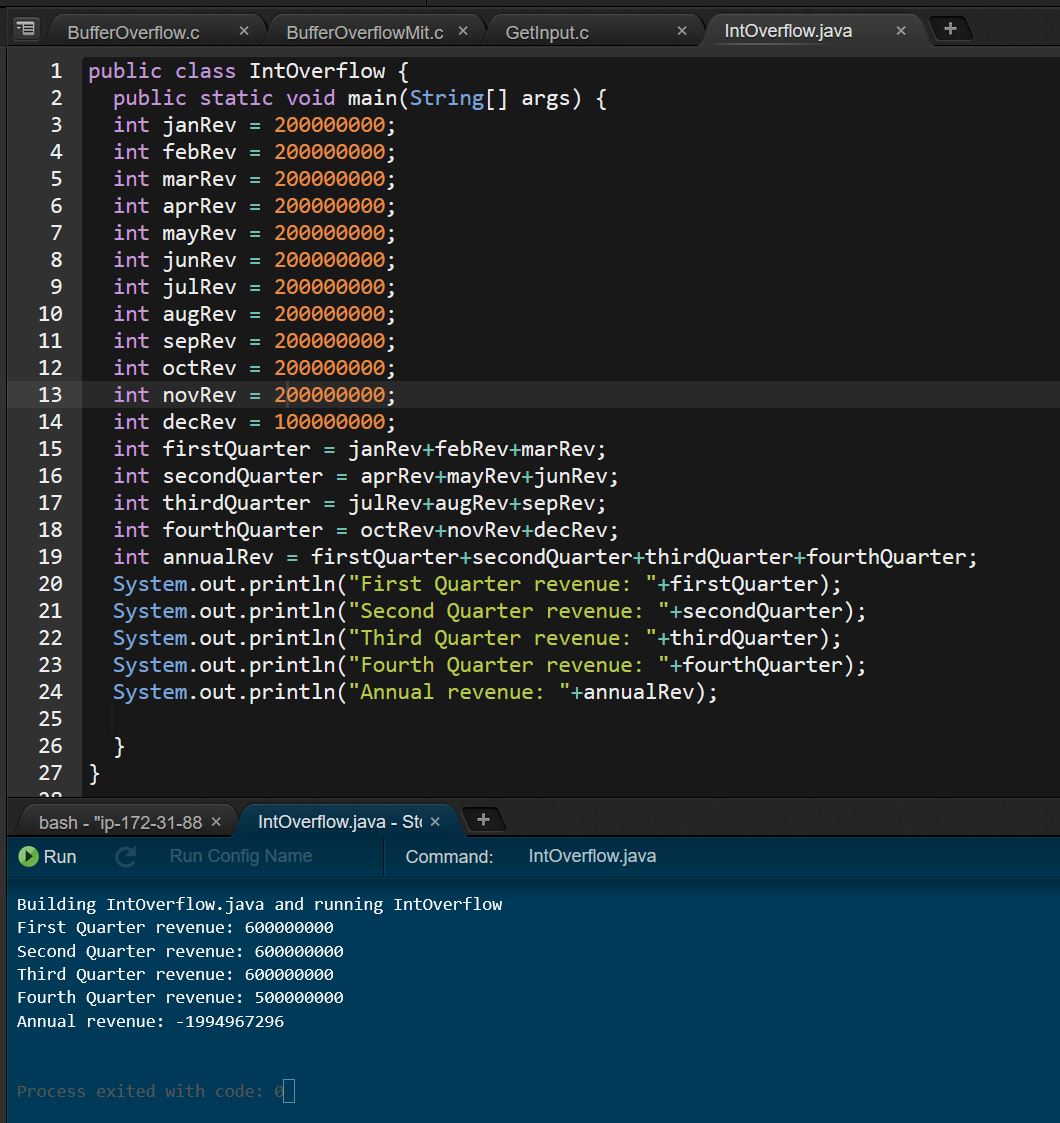
**Example #2**

**Example of Program Vulnerable to Integer Overflow or Wraparound**

****

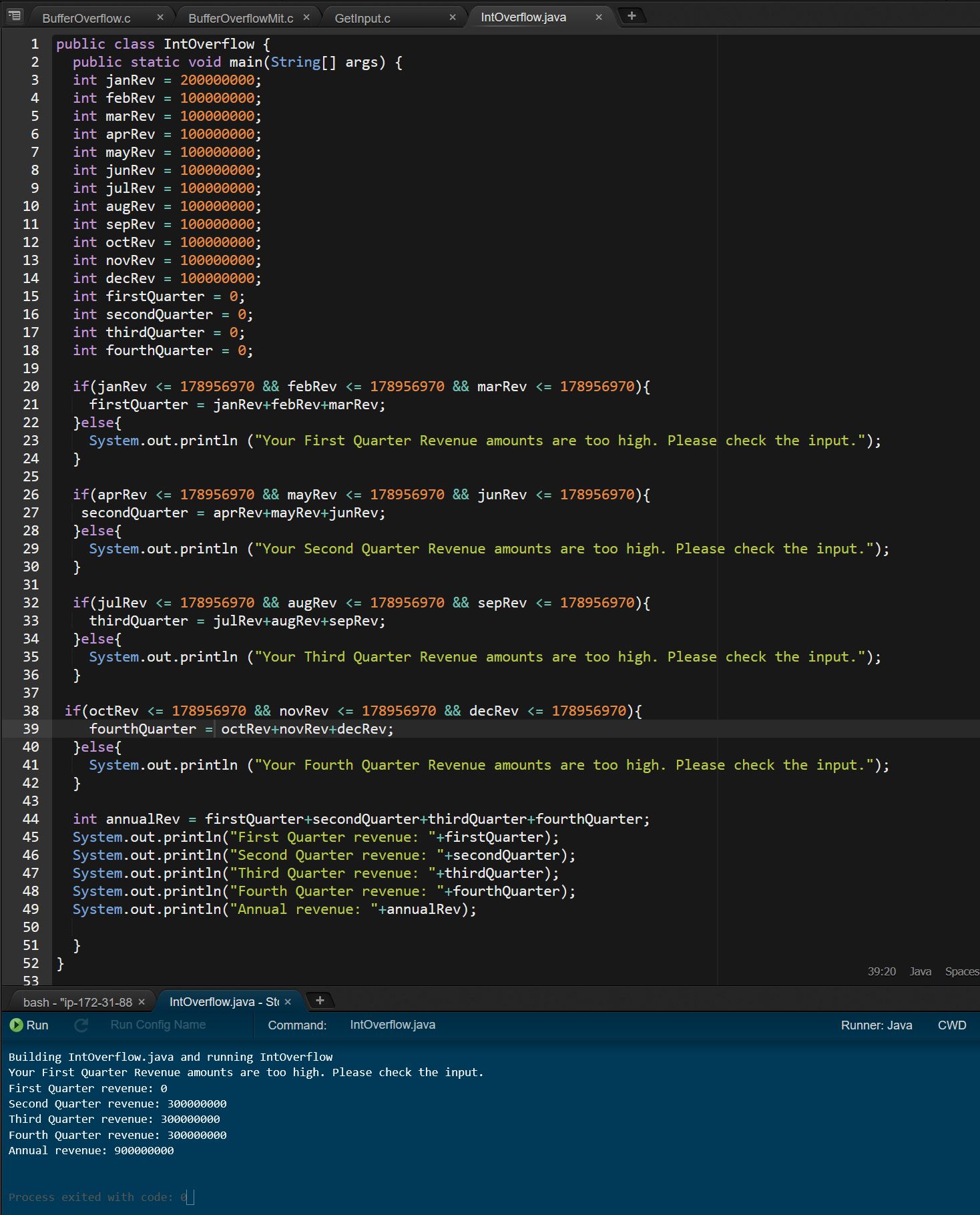
This program is vulnerable to integer overflow because there are no checks to ensure the variables used to calculate quarterly and annual revenue are of the appropriate size and that the resulting number will be able to be held within the assigned variables. In Java an int can hold a maximum possible value of 2,147,483,647. The program above calculates annual revenue to be 2,100,000,000. A slight increase in revenue will cause an integer wraparound.

**Example of Integer Wraparound**

****

Ways to mitigate this include changing the variable type to something that could hold larger values. In this case a **long** type could how values up to 9,223,372,036,854,775,808. However, that wouldn’t solve problems if somewhere bad data was introduced into the program. A better solution would be to include checks within the program to ensure the variable values fall within a predetermined range.

**Example of Input Validation**



Here we’ve created if/else statements to validate the size of the variables before any calculations are done with the data. This ensures that no value can be of a size that would cause the Annual Revenue calculation to integer overflow.