Week 4 Homework 3

Matt Rieser

SDEV 325 – Section 6381  
October 14, 2018

Professor Zachary Fair

**Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**

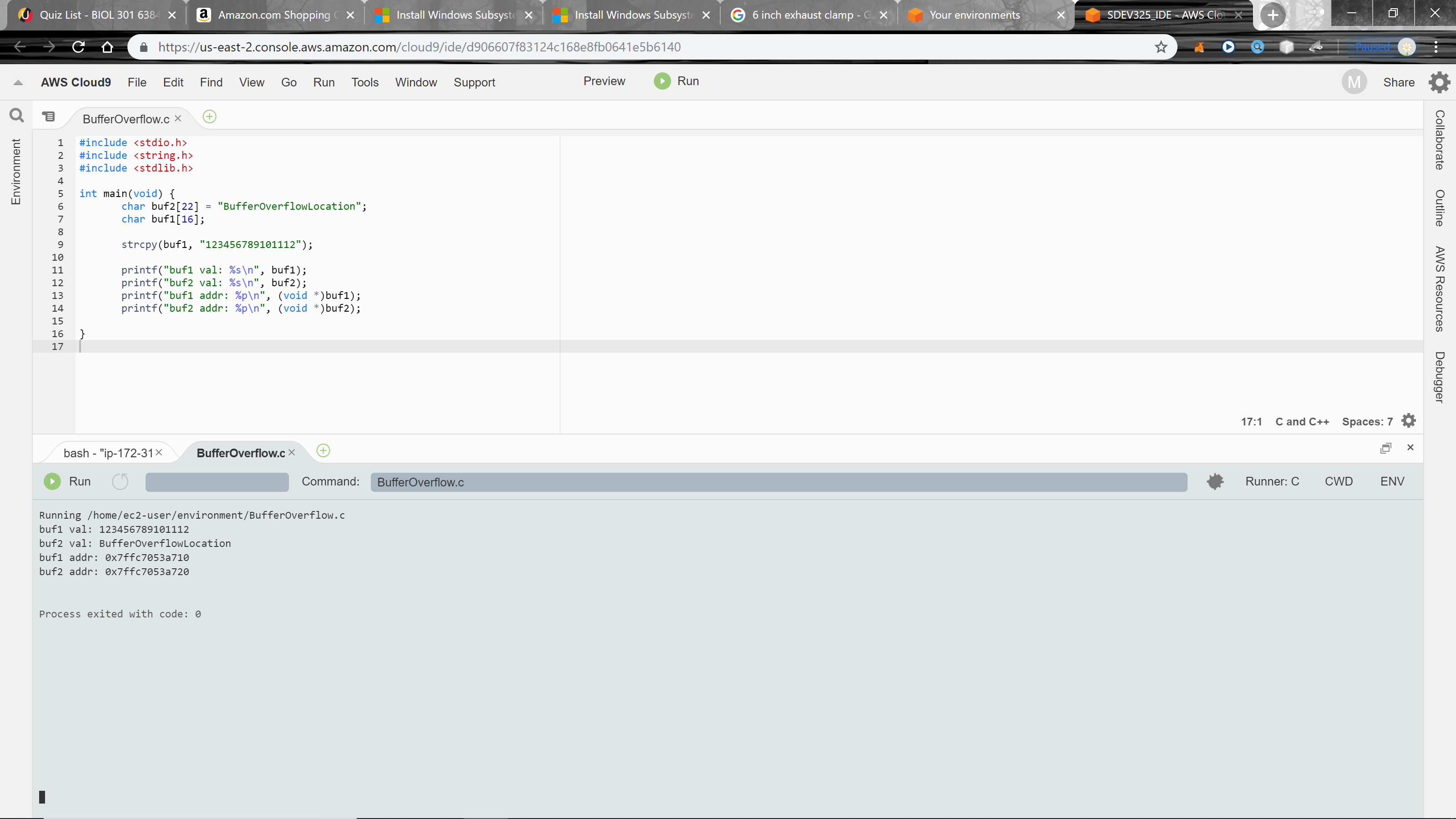
**Description**

The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

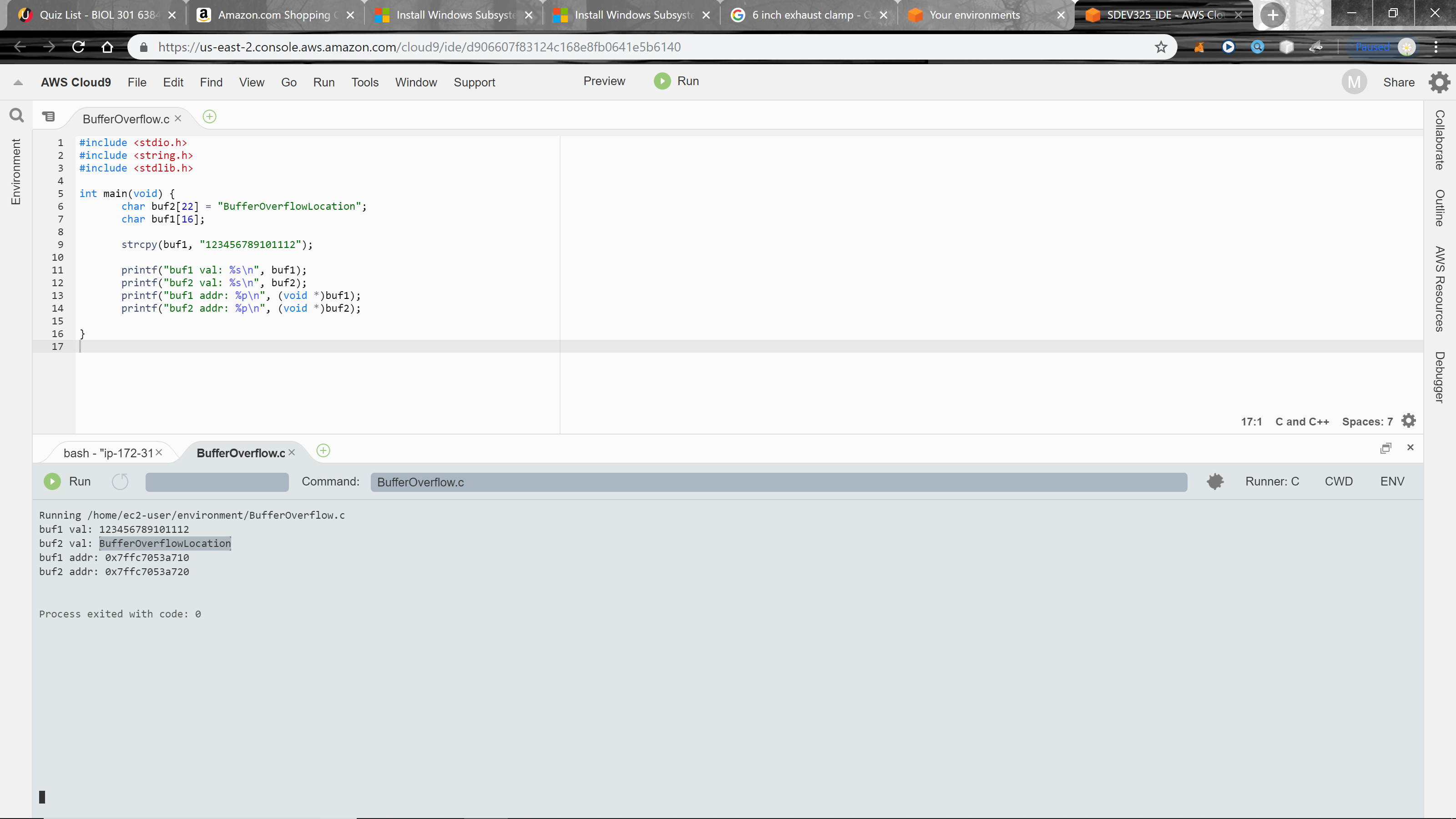
**Extended Description**

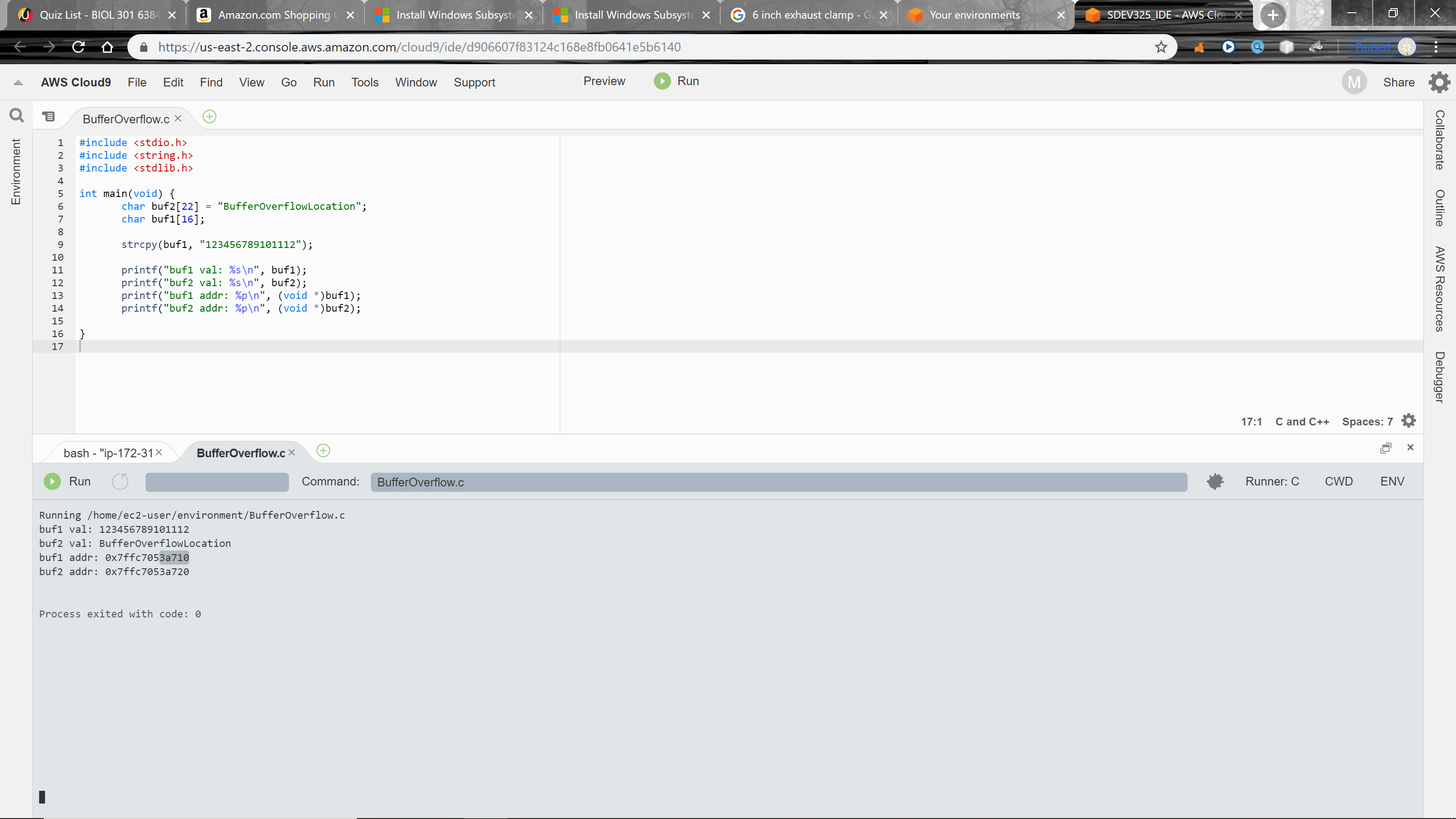
A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold, or when a program attempts to put data in a memory area outside of the boundaries of a buffer. The simplest type of error, and the most common cause of buffer overflows, is the "classic" case in which the program copies the buffer without restricting how much is copied. Other variants exist, but the existence of a classic overflow strongly suggests that the programmer is not considering even the most basic of security protections.

Resource – <http://cwe.mitre.org/top25/index.html#CWE-120>

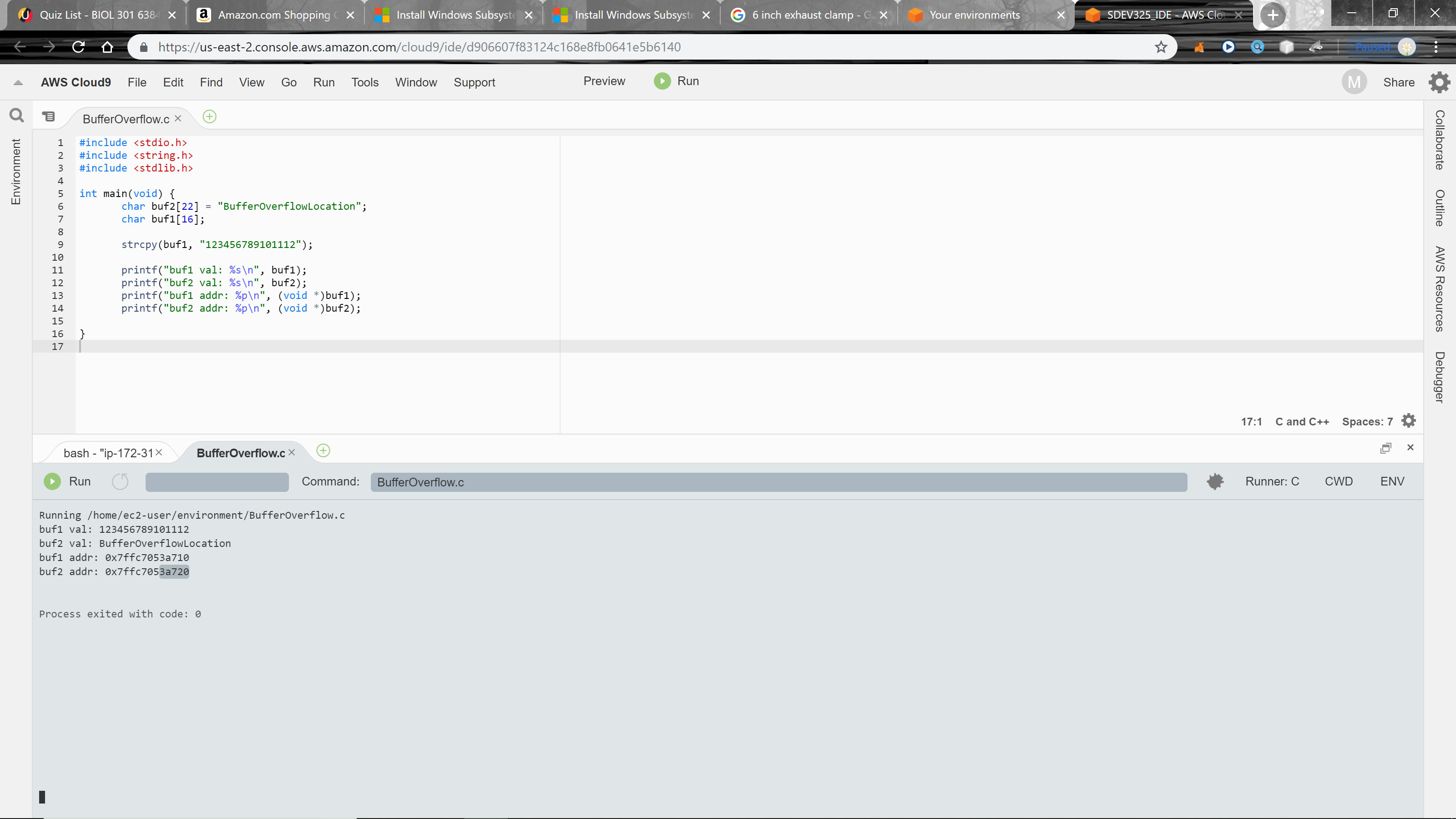


Above – Shows buf1 val only having 15 characters



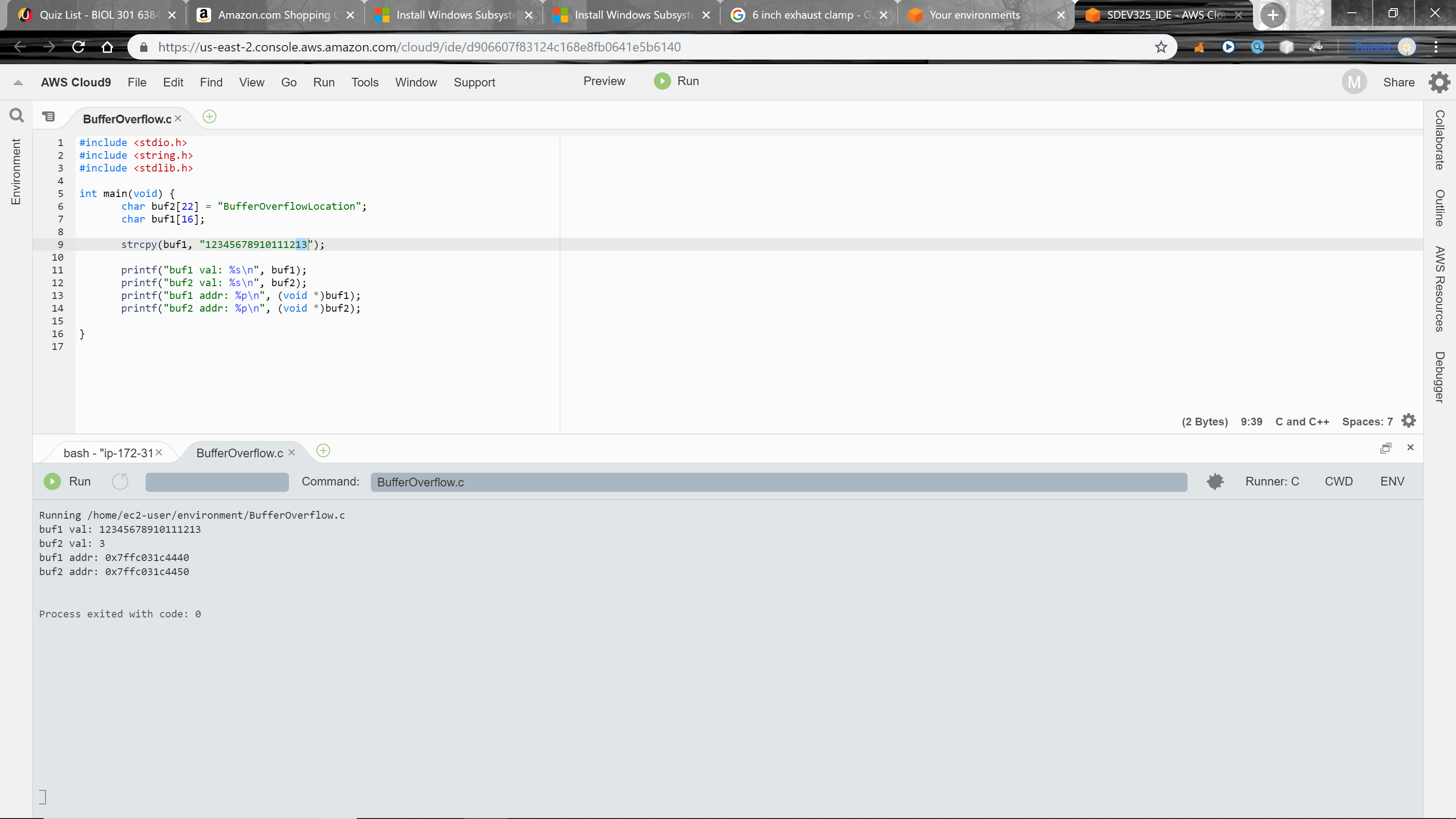
Above – shows the location where the buffer will overflow into after the 16 characters limit is reached

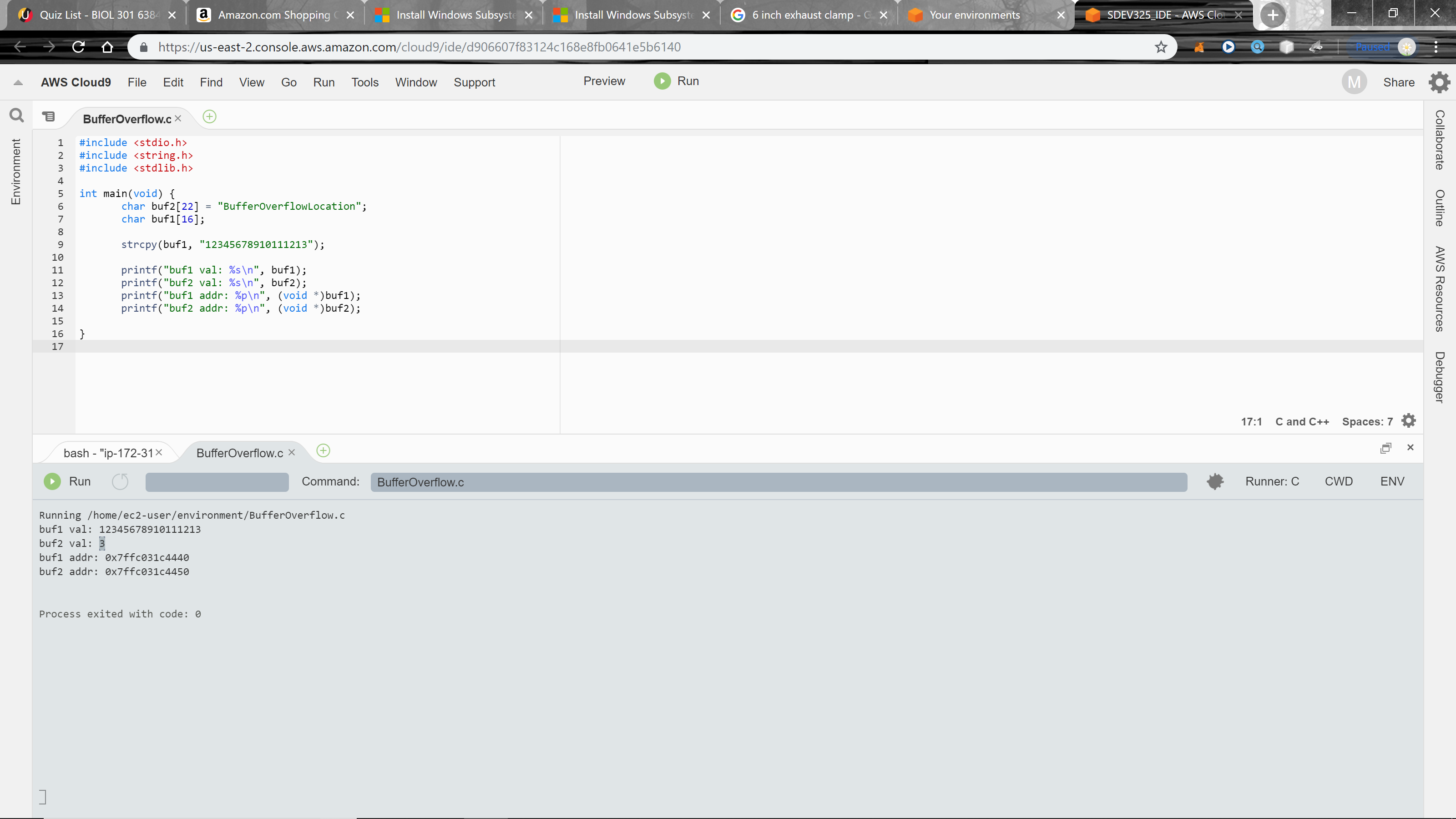
Above – notice the location of buf1 address ending is 3a710



Above – notice buf2 address ending is 3a720

Both addresses are located next to each other, 3a710 and next is 3a720 and next would be 3a730 and so on. So, when an overflow of characters occurs in 3a710, the overflow would show up in 3a720. If the character amount exceeds buf2 character amount of 16, then it would also overflow into buffer3 … if there was one in this situation.

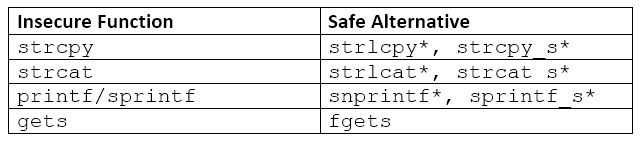
**Buffer Overflow**

Above – notice the added 13 to buf1

Above – notice that the 16-character limit was reached and the next character after the 16th byte was overflowed in the next buffer location. In this case the next location is buf2 and the 3 was displayed.

**Buffer Overflow Fix**

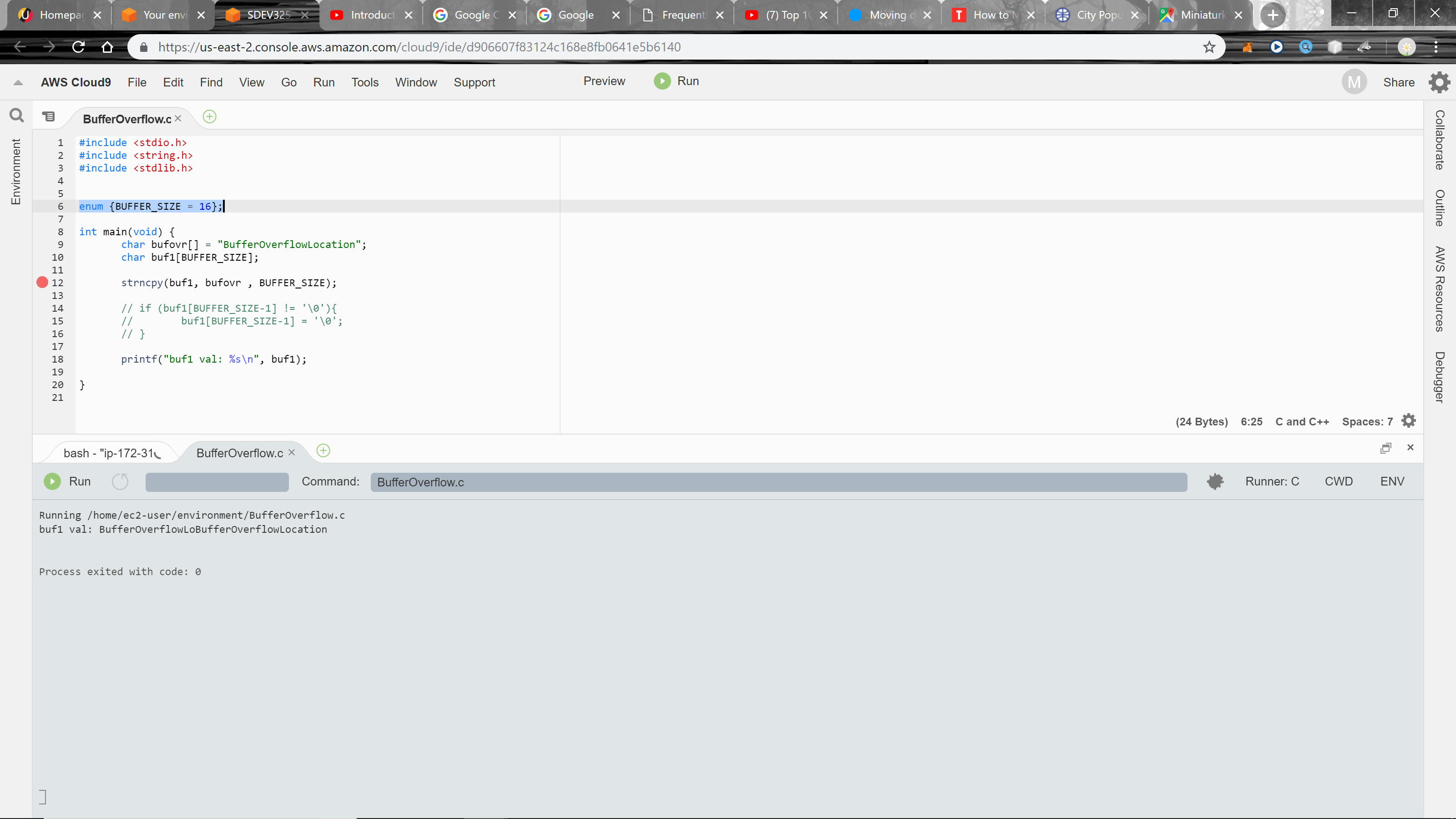
First off creating a buffer overflow (segmentation fault) is quite difficult given that most languages don’t have direct access to the buffer. C/C++ however are susceptible to a buffer overflow attack. C more so because there isn’t that many ways to fix it. C has alternatives, but most are not built into their standard libraries:



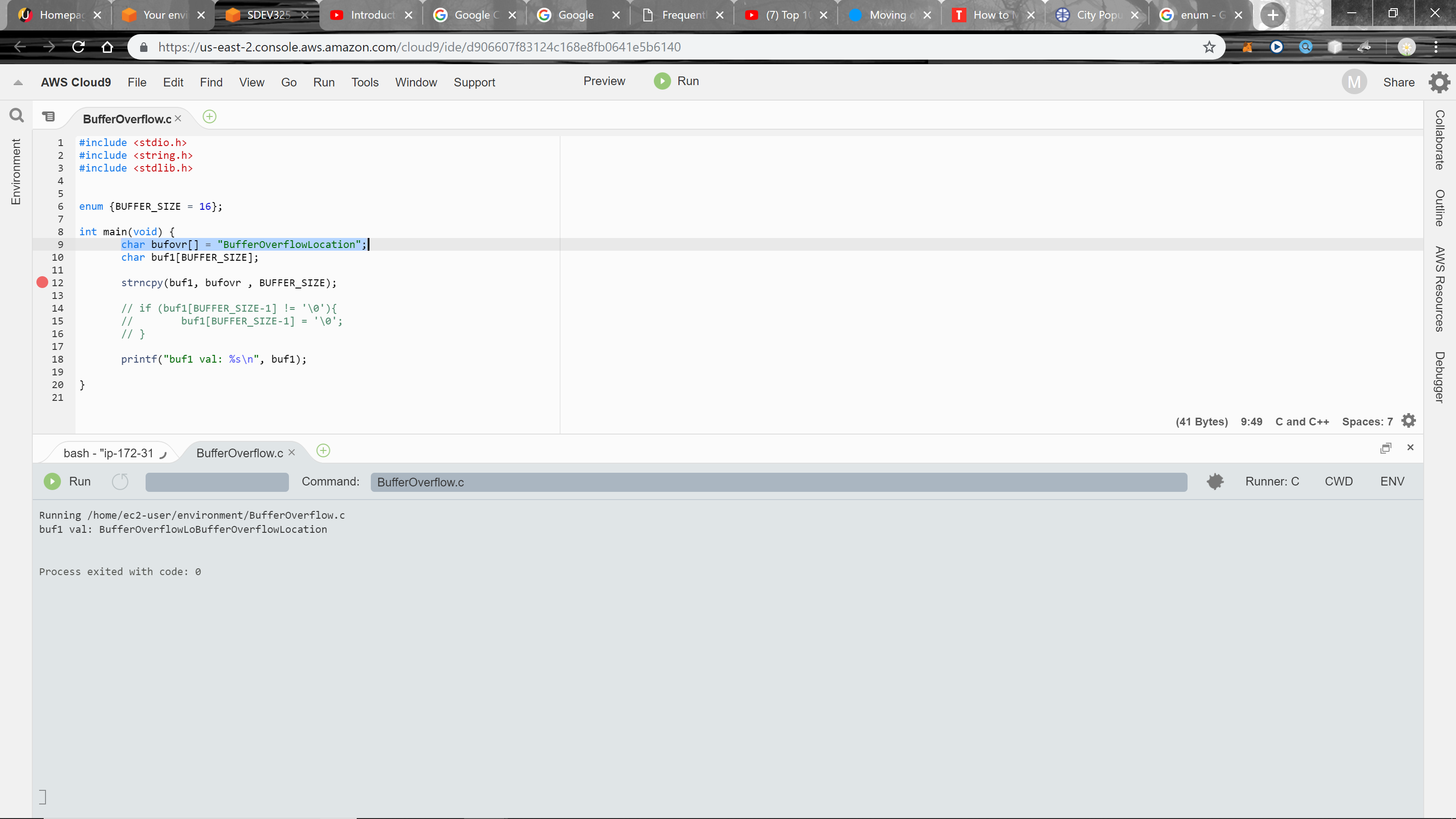
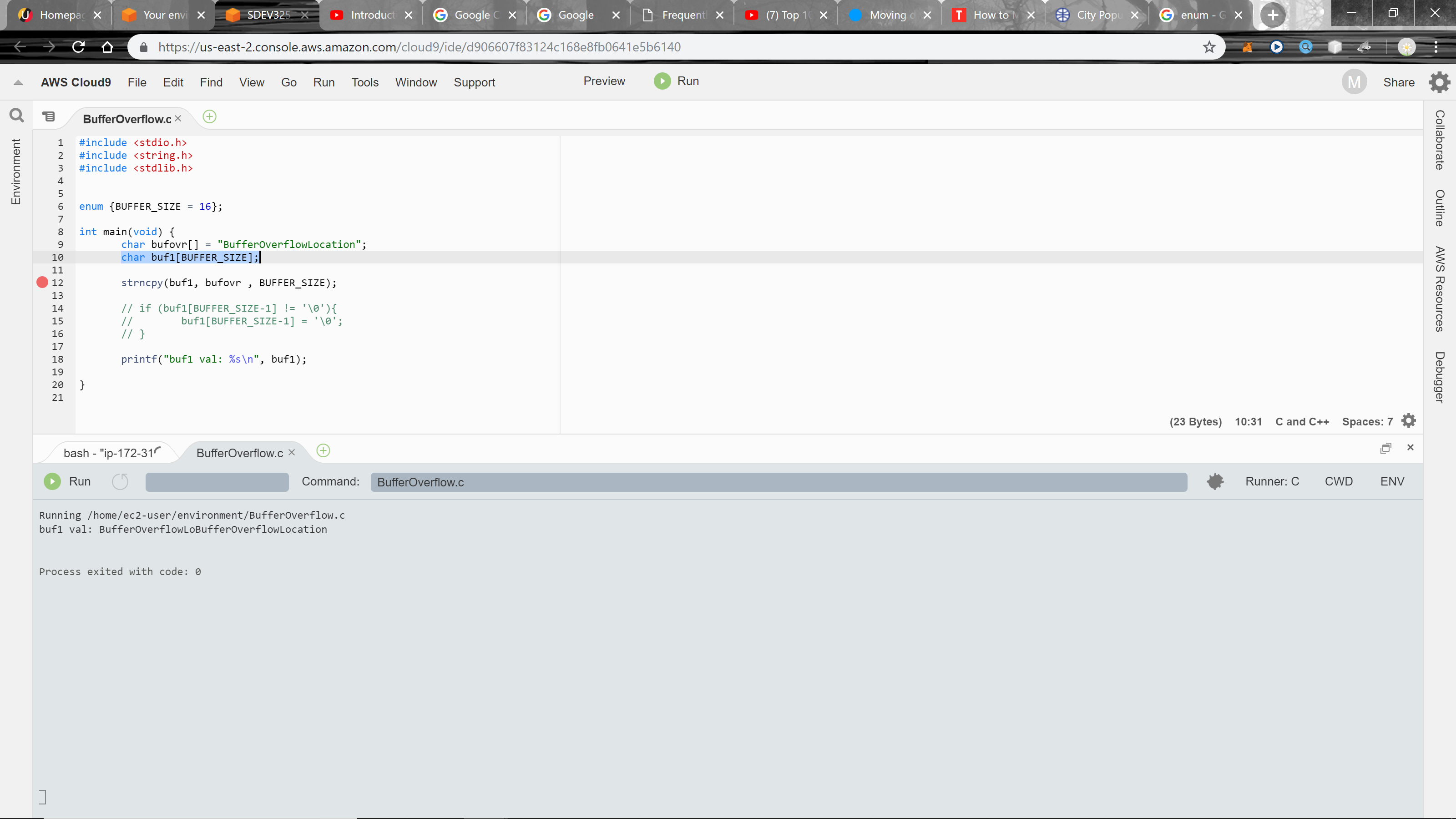
\*Asterisks denote functions that are not part of C Standard Libraries.

So, if you have used any of the following insecure functions all you have to do is change it to a safe one. My insecure code was:

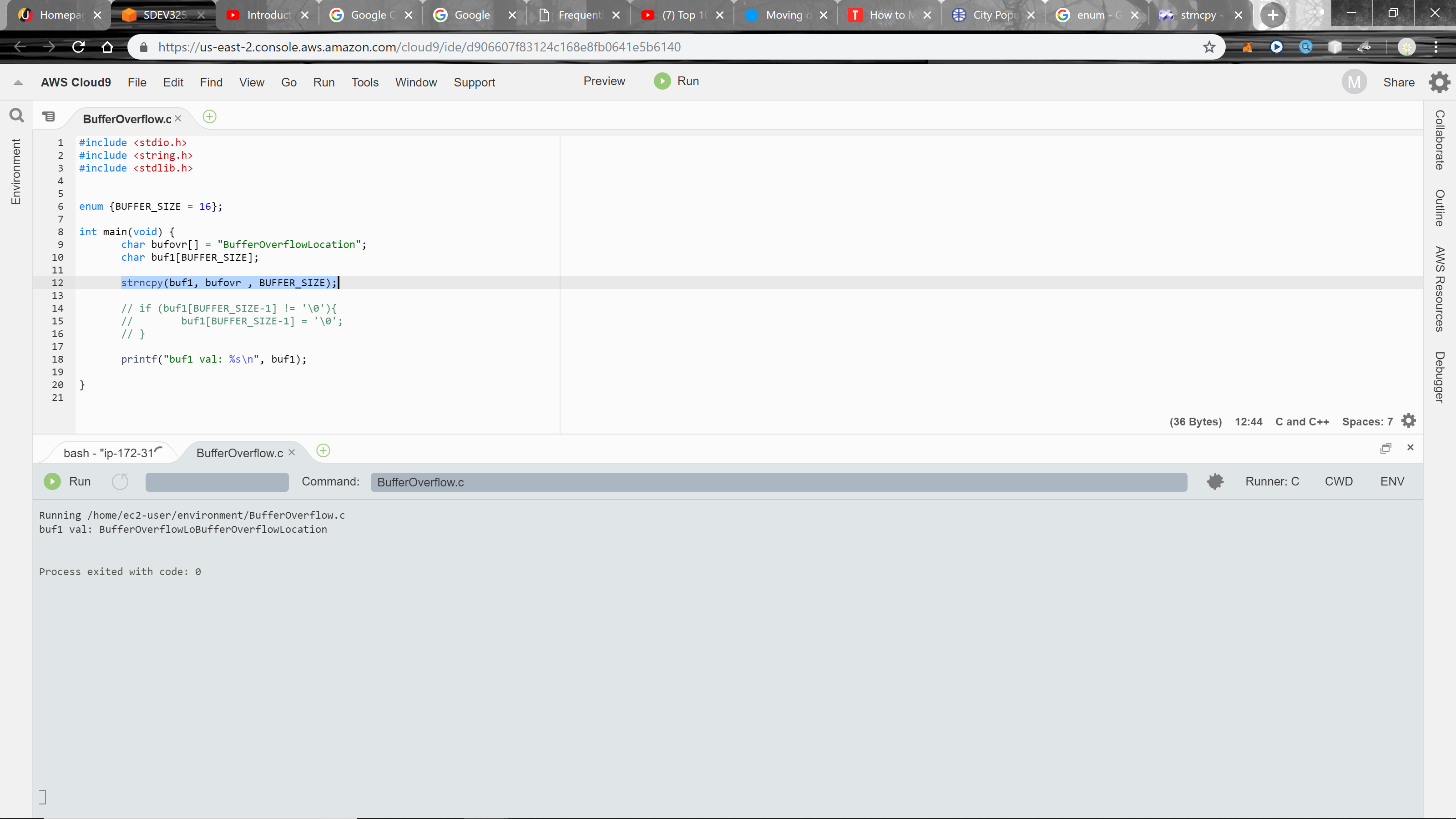
strcpy(buf1, "12345678910111213");

The fix I found was to create check using an if statement. I am going to walk through the new code.

Above – the enumeration of BUFFER\_SIZE limit set to 16 bytes.

Above – my bufover (bufferoverflow test)

Above – buf1 is set to the constraint size of 16 bytes

Above – buf1 is the destination, bufovr is the source, and [BUFFER\_SIZE] is the maximum number of characters

destination

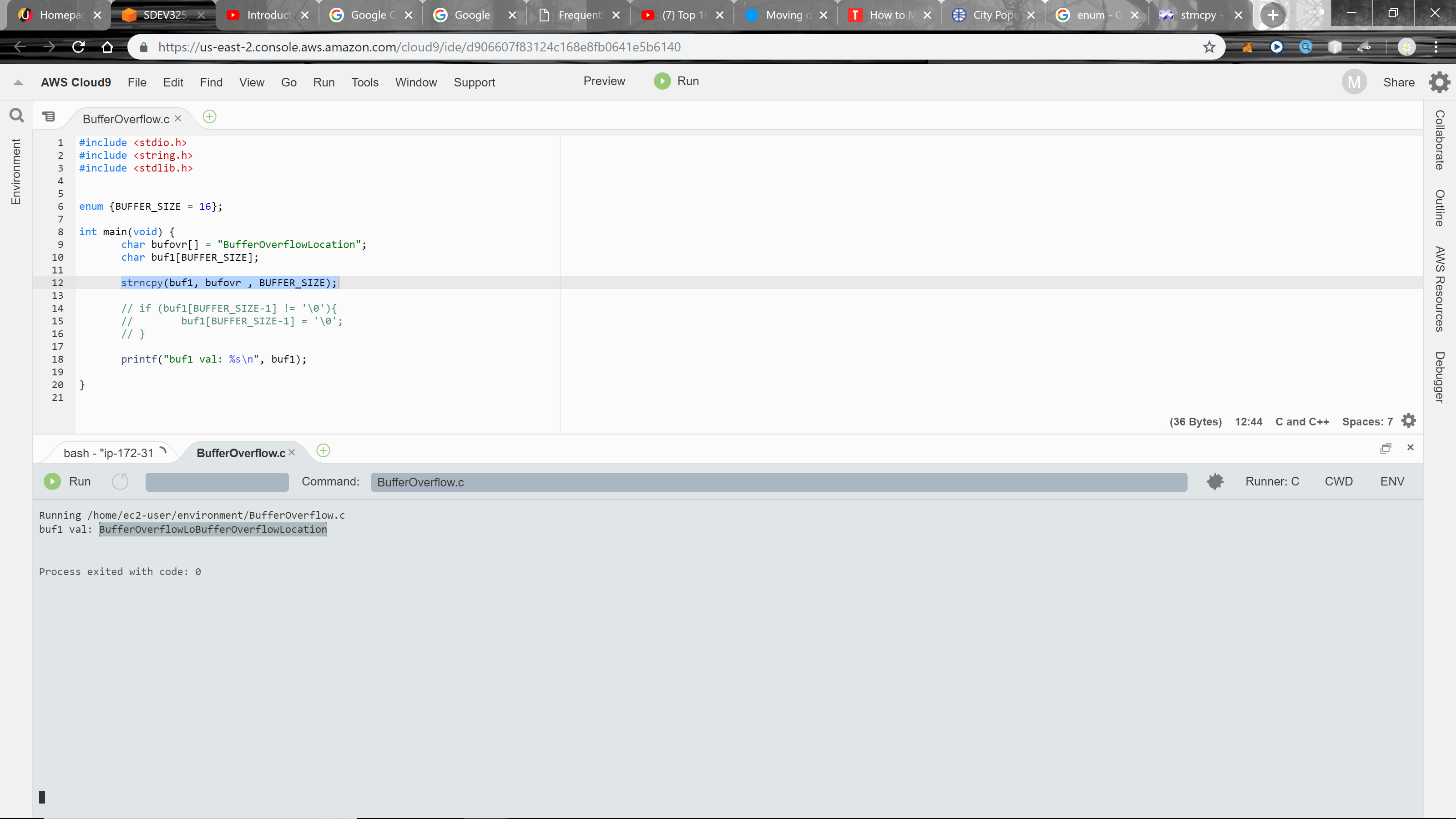
Pointer to the destination array where the content is to be copied.

source

C string to be copied.

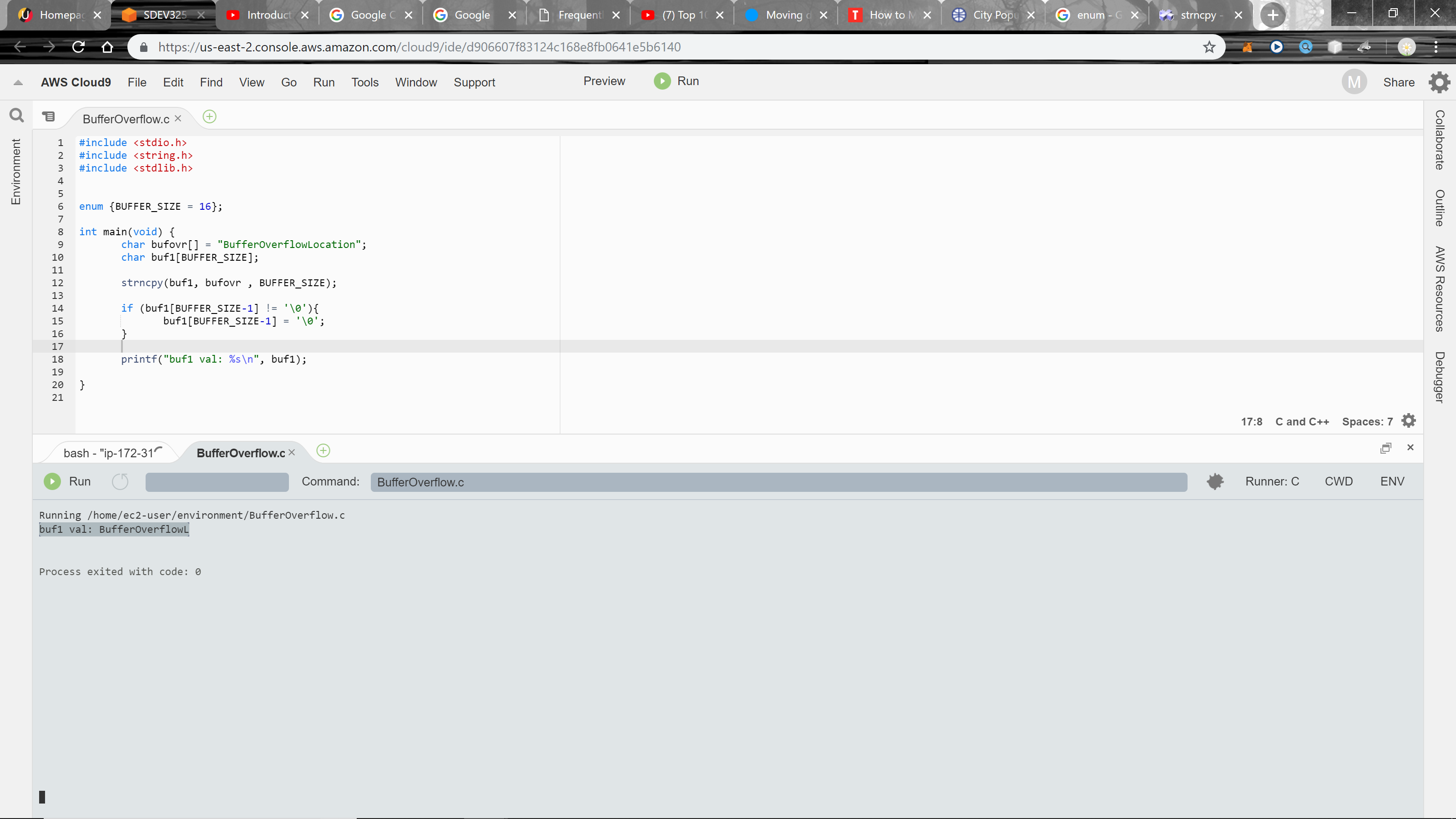
num

Maximum number of characters to be copied from *source*.  
[size\_t](http://www.cplusplus.com/cstring:size_t) is an unsigned integral type.

Output :

buf1 val: BufferOverflowLoBufferOverflowLocation

Above – notice the output prints the first 16 characters of BufferOverflowLo. And then after it also overflows the output with the full, BufferOverflowLocation. The fix is an if statement that uses the BUFFER\_SIZE limit that will only print the specified amount of characters/bytes.

Above – notice the output is now BufferOverflowL. The bufferoverflow check was successful. It printed 15 characters of the bufovr string and then once it got to the 16 character/byte it ended the process.

**Use of Externally-Controlled Format String**

**Description**

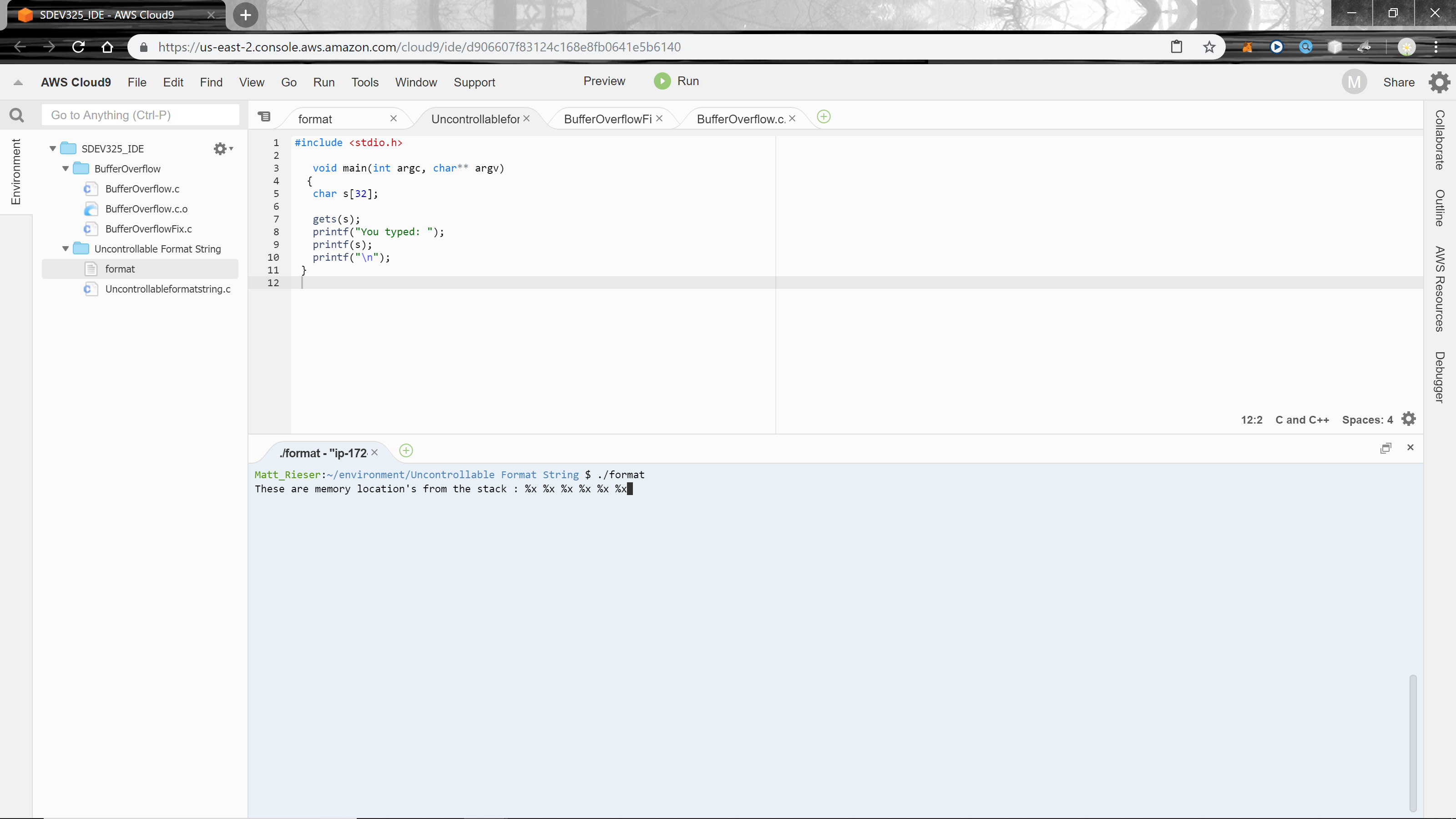
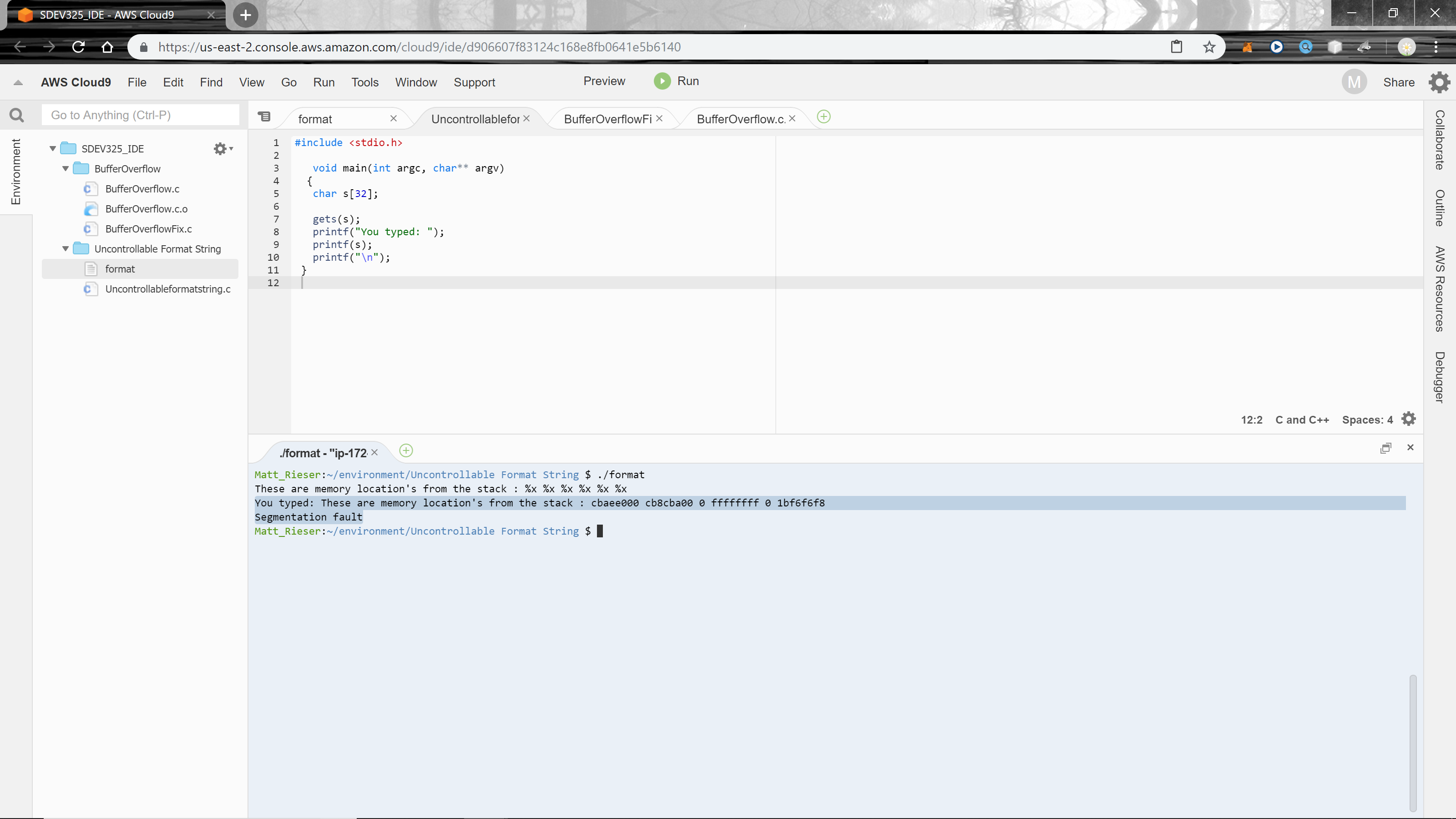
The software uses a function that accepts a format string as an argument, but the format string originates from an external source.

**Extended Description**

When an attacker can modify an externally-controlled format string, this can lead to buffer overflows, denial of service, or data representation problems.

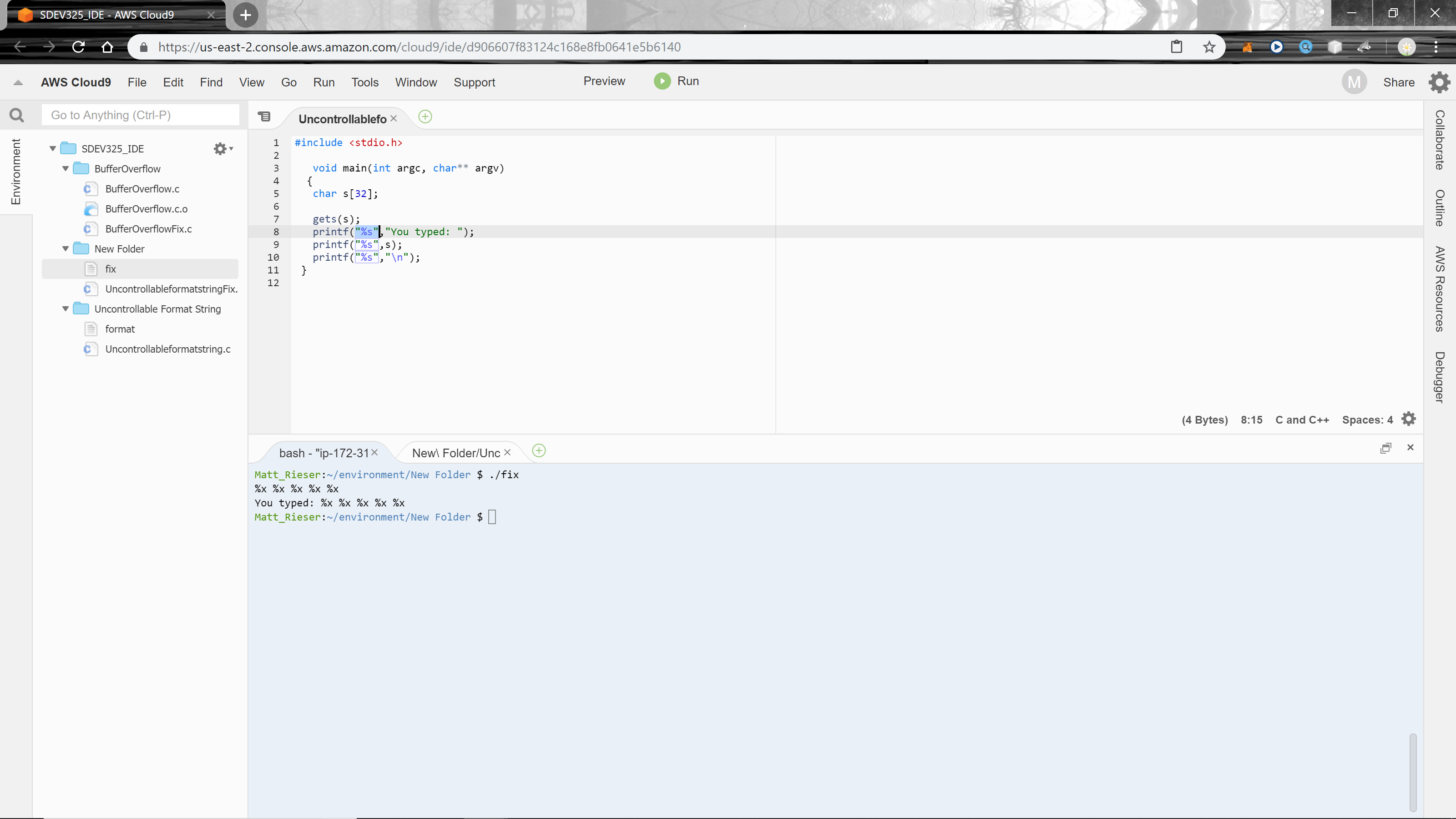
It should be noted that in some circumstances, such as internationalization, the set of format strings is externally controlled by design. If the source of these format strings is trusted (e.g. only contained in library files that are only modifiable by the system administrator), then the external control might not itself pose a vulnerability.

Resource- <http://cwe.mitre.org/data/definitions/134.html>

Above – user input to so format string vulnerability.

Above - Notice the string format %x outputs the stack memory addresses: cbaee000 cb8cba00 0 ffffffff 0 1bf6f6f8. It also seems like the input %x caused a segmentation fault (when a program attempts to access memory it has either not been assigned by the operating system or is otherwise not allowed to access.)

**Uncontrollable Format String Fix**

**Potential Mitigations**

|  |
| --- |
| **Phase: Requirements**  Choose a language that is not subject to this flaw. |
| **Phase: Implementation**  Ensure that all format string functions are passed a static string which cannot be controlled by the user and that the proper number of arguments are always sent to that function as well. If at all possible, use functions that do not support the %n operator in format strings. [[REF-116](http://cwe.mitre.org/data/definitions/134.html#REF-116)] [[REF-117](http://cwe.mitre.org/data/definitions/134.html#REF-117)] |
| **Phase: Build and Compilation**  Heed the warnings of compilers and linkers, since they may alert you to improper usage. |

Above – In my case the addition of “%s” to each printf statement did the trick.

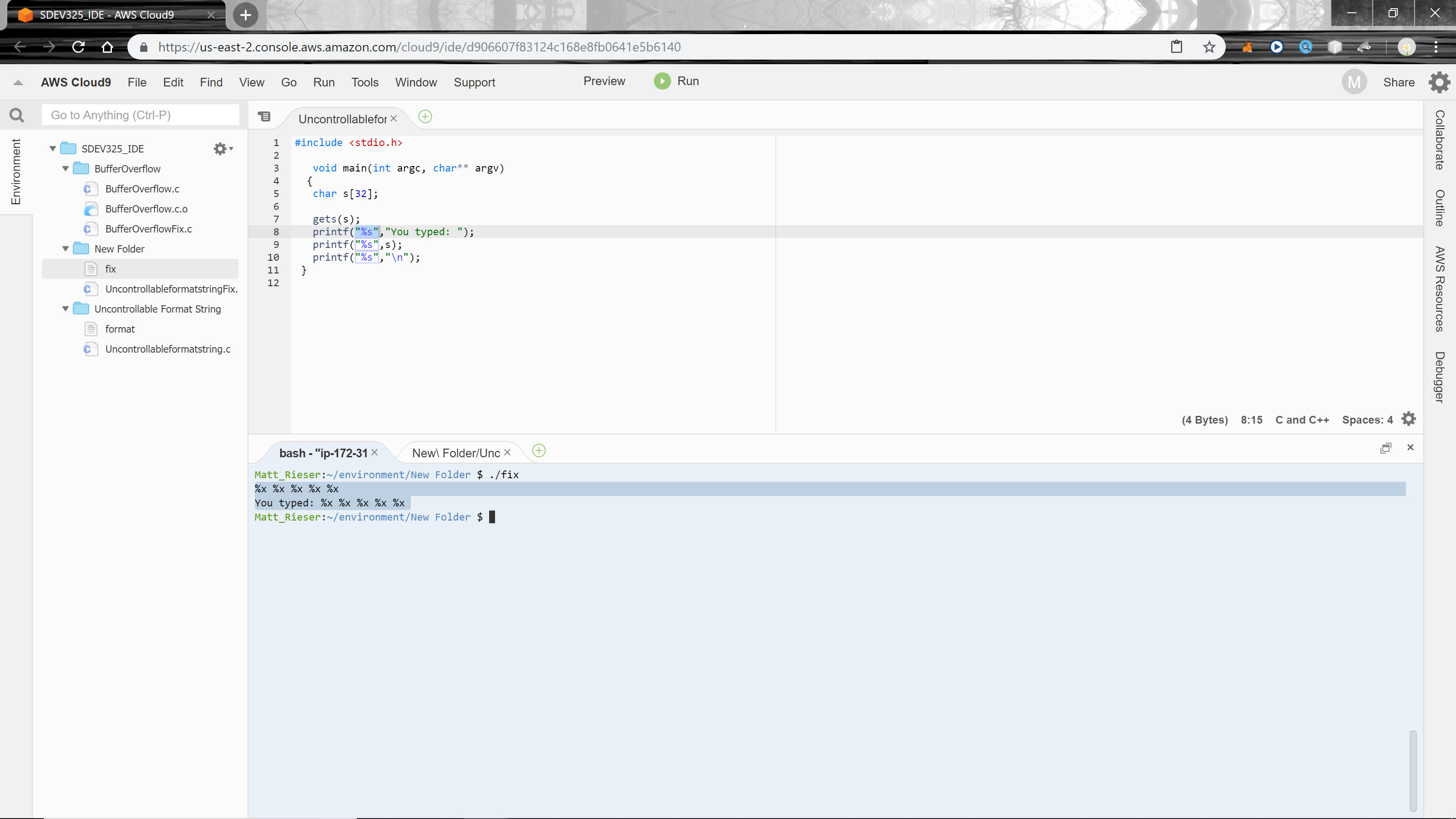
So, the rule of thumb in preventing format string bugs to be exploited by hackers is not to use a non-constant as a format string argument in all Printf family of functions. Instead, the correct usage of Printf without non-constant should be as following;

printf(user\_supplied\_data); // Prone to Vulnerable

printf("%s", user\_supplied\_data); // Correct Usage

fprintf(stderr, user\_supplied\_data); // Prone to Vulnerable

fprintf(stderr, "%s", user\_supplied\_data); // Correct Usage

Resource - <https://resources.infosecinstitute.com/format-string-bug-exploration/#gref>

Above – notice my input was %x and the result was the same (not like before with the stack memory addresses displayed).