Codio Activity - Exploring Python tools and features

Part I

In this example, you will compile and run a program in C using the **Codio workspace** provided (Buffer Overflow in C). The program is already provided as bufoverflow.c - a simple program that creates a buffer, asks you for a name and prints it back to the screen.

This is the code in bufoverflow.c (also available in the Codio workspace):

```
#include <stdio.h>
int main(int argc, char **argv)
{
char buf[8]; // buffer for eight characters
printf("enter name:");
gets(buf); // read from stdio (sensitive function!)
printf("%s\n", buf); // print out data stored in buf
return 0; // 0 as return value
{
```

Now, compile and run the code. To test it, enter your first name (or at least the first 8 characters of it); you should get the output of just your name repeated back to you.

Run the code a second time (from the command window; this can be achieved by entering ./bufoverflow on the command line). This time, enter a string of 10 or more characters.

1. What happens?
2. What does the output message mean?

Output after running the code:

```
codio@augustjustice-sharonstamp:~/workspace$ gcc bufoverflow.c -o bufoverflow && ./bufoverflow
bufoverflow.c: In function ‘main’:
bufoverflow.c:8:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-
function-declaration]
    gets(buf);              // read from stdio (sensitive function!)
    ^~~~
    fgets
/tmp/ccnxtD3J.o: In function `main':
```

```
bufoverflow.c:(.text+0x3c): warning: the `gets' function is dangerous and should not be used.
Enter name: Hainadin
Hainadin

codio@augustjustice-sharonstamp:~/workspace$ gcc bufoverflow.c -o bufoverflow && ./bufoverflow
bufoverflow.c: In function ‘main’:
bufoverflow.c:8:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
    gets(buf);          // read from stdio (sensitive function!)
    ^~~~
    fgets
/tmp/ccQjNn6c.o: In function `main':
bufoverflow.c:(.text+0x3c): warning: the `gets' function is dangerous and should not be used.
Enter name: HainadineC
HainadineC
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Some functions are dangerous as they lack security measures. Examples include gets() and the >> operator, both of which do not perform bounds checking and can easily be exploited by attackers to overflow the destination buffer (CWE, N.D.). For instance, Sharma (2022) wrote that the gets() function in C solves this issue by accepting a size limit and preventing overflows.

When data is written to a buffer that is too small to hold it, a buffer overflow occurs. This overwrites adjacent memory addresses and can be avoided by implementing proper bounds checking (Cobb, 2021). Buffer overflows can corrupt a program's memory, resulting in unpredictable behaviour or program crashes. The operating system's memory protection mechanisms detect these violations and terminate the program to prevent security risks and data corruption (Welekwe, 2020).

The compiler generates a "stack smashing detected" error to defend against buffer overflows caused by input exceeding buffer capacity. C code with a buffer capacity of eight can cause this error if exceeded (Educative, N.D.).

1. **What happens?**

Providing a string of 10 or more characters to the code will cause a buffer overflow. The `buf` array is only 8 characters, and the `gets()` function does not check the input size, leading to memory overwriting. This can cause crashes, unintended behaviour, or exploitation.

2. **What does the output message mean?**

The output message is a fault error. This happened because the buffer overflow corrupts the program's memory, leading to unpredictable behaviour. The operating system's memory protection mechanisms detect this violation and terminate the program to prevent potential security risks or data corruption.

Part II

Now, carry out a comparison of this code with one in Python (Buffer Overflow in Python), following these instructions:

In the Codio workspace, you will be using the file called Overflow.py:

```
buffer=[None]*10
for i in range (0,11):
   buffer[i]=7
print(buffer)
```

Run your code using Python overflow.py (or use the codio rocket icon).

1. What is the result?

- Read about Pylint at http://pylint.pycqa.org/en/latest/tutorial.html
- Install pylint using the following commands:
  pip install pylint (in the command shell/ interpreter)
- Run pylint on your Overflow.py file and evaluate the output:
  pylint Overflow.py

- (Make sure you are in the directory where your file is located before running Pylint)

2. What is the result? Does this tell you how to fix the error above?

Output after running the code:

```
codio@gridbishop-specialnumber:~/workspace$ python3 Overflow.py
Traceback (most recent call last):
  File "Overflow.py", line 3, in <module>
    buffer[i]=7
IndexError: list assignment index out of range
```

1. Running the Python code results in an " IndexError: list assignment index out of range". This is because, in the code, the buffer has 10 elements, but the loop attempts to write through 15 elements, which results in an error. When the count reaches 10, trying to assign buffer[10] = 7 will result in an "IndexError" since index 10 is out of bounds for the buffer list.

2. **Output after running the code:**

```
pylint Overflow.py
```

```
codio@gridbishop-specialnumber:~/workspace$ pylint Overflow.py
************* Module Overflow
Overflow.py:4:0: C0303: Trailing whitespace (trailing-whitespace)
Overflow.py:5:0: C0304: Final newline missing (missing-final-newline)
Overflow.py:1:0: C0103: Module name "Overflow" doesn't conform to snake_case naming style
(invalid-name)
Overflow.py:1:0: C0114: Missing module docstring (missing-module-docstring)

------------------------------------------------------------------
Your code has been rated at 0.00/10 (previous run: 0.00/10, +0.00)
```

I have found four errors in the code. To retrieve the solution for each error, I can run the following command according to the explanation of Pylint 3.0.0a8-dev0 documentation (N.D.): "pylint --help-msg=" and add the message information between parentheses.

```
codio@gridbishop-specialnumber:~/workspace$ pylint --help-msg=missing-module-docstring
:missing-module-docstring (C0114): *Missing module docstring*
```

```
  Used when a module has no docstring. Empty modules do not require a docstring.
  This message belongs to the basic checker.

codio@gridbishop-specialnumber:~/workspace$ pylint --help-msg=trailing-whitespace
:trailing-whitespace (C0303): *Trailing whitespace*
  Used when there is whitespace between the end of a line and the newline. This
  message belongs to the format checker.

codio@gridbishop-specialnumber:~/workspace$ pylint --help-msg=trailing-whitespace
:trailing-whitespace (C0303): *Trailing whitespace*
  Used when there is whitespace between the end of a line and the newline. This
  message belongs to the format checker.

codio@gridbishop-specialnumber:~/workspace$ pylint --help-msg=invalid-name
:invalid-name (C0103): *%s name "%s" doesn't conform to %s*
  Used when the name doesn't conform to naming rules associated to its type
  (constant, variable, class...). This message belongs to the basic checker.
```

Codio Activity: The Producer-Consumer Mechanism

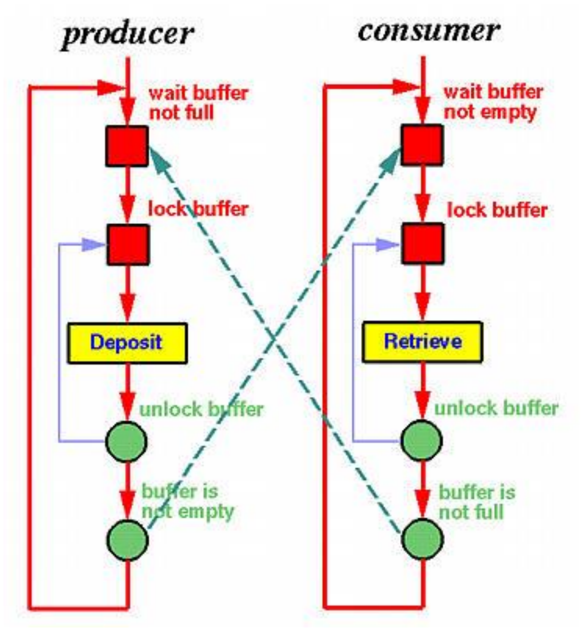
Producer/Consumer Problem (also known as the ‘bounded buffer’ problem):

- A ‘producer’ produces items at a particular (unknown and sometimes unpredictable) rate.
- A ‘consumer’ is consuming the items – again, at some rate.

For example, a producer-consumer scenario models an application producing a listing that a printer process must consume. A keyboard handler creates a data line that an application program will consume. This is shown in the picture below (Shene, 2014).

Items are placed in a buffer when produced, so:

- Consumers should wait if there isn’t an item to consume
- Producer shouldn’t ‘overwrite’ an item in the buffer



Synchronisation is necessary because:

- If the consumer has not taken out the current value in the buffer, then the producer should not replace it with another.
- Similarly, the consumer should not consume the same value twice.

Task

Run producer-consumer.py in the provided Codio workspace (**Producer-Consumer Mechanism**), where the queue data structure is used.

A copy of the code is available here for you.

```
# code source: https://techmonger.github.io/55/producer-consumer-python/

from threading import Thread
from queue import Queue

q = Queue()
final_results = []

def producer():
    for i in range(100):
        q.put(i)


def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        final_results.append(result)
        q.task_done()


for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print (final_results)
```

Answer the following questions:

1. How is the queue data structure used to achieve the purpose of the code?
2. What is the purpose of q.put(I)?
3. What is achieved by q.get()?
4. What functionality is provided by q.join()?
5. Extend this producer-consumer code to make the producer-consumer scenario available securely. What technique(s) would be appropriate to apply?

1. **How is the queue data structure used to achieve the purpose of the code?**

Using the queue data structure enables the implementation of the producer-consumer mechanism (Tech Monger, N.D). In this case, the consumer thread retrieves items from the queue, which acts as a buffer for the producer's goods until they are ready for use by the customer.

2. **What is the purpose of q.put(I)?**

In this programme, the producer adds items (numbers ranging from 0 to 99) to the queue using the q.put(i) method. A new item is added to the queue on each iteration of the producer loop to imitate the production of data that needs to be processed.

3. **What is achieved by q.get()?**

The consumer threads access entries from the queue using q.get(). Every consumer thread in this code reliably takes items from the queue, squares the result, and stores the result in the list labelled "final_results."

4. **What functionality is provided by q.join()?**

The q.join() function waits for all items in the queue to be handled and designated as complete by q.task_done(), ensuring that the program won't progress until all items are processed.

5. **Extend this producer-consumer code to make the producer-consumer scenario available securely. What technique(s) would be appropriate to apply?**

Thread synchronisation prevents multiple processes or threads from simultaneously executing the same critical section. Synchronisation techniques control access to the area, avoiding race conditions and unpredictable variable values (GeeksforGeeks, 2017).

Employing thread synchronisation and locking techniques guarantees the threads' safety when numerous threads deal with shared resources, such as a queue. To ensure safe access to shared data, mutexes (locks) can be implemented (GeeksforGeeks, 2022). The following code has been updated to increase security based on concepts from Python Tutorial (N.D.):

```
from threading import Thread, Lock
from queue import Queue

q = Queue()
final_results = []
lock = Lock()  # Create a lock for shared data protection

def producer():
    for i in range(100):
        q.put(i)

def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        with lock:  # Acquire the lock before modifying shared data
            final_results.append(result)

        q.task_done()

for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print(final_results)
```

To ensure secure access to the "final_results" list, a lock has been added to the code that consumer threads use to append results. This "lock" prevents unauthorised access. By using the "with lock" statement, potential race conditions are avoided, and thread safety is promoted. This limits access to shared data to one thread at a time.

References:

Sharma, T. (2022). *gets() Function in C*. [online] Scaler Topics. Available at: https://www.scaler.com/topics/gets-in-c/ [Accessed 27 Aug. 2023].

CWE (N.D.). *CWE - CWE-242: Use of Inherently Dangerous Function (4.12)*. [online] Available at: https://cwe.mitre.org/data/definitions/242.html#:~:text=The%20gets()%20function%20is [Accessed 27 Aug. 2023].

Cobb, M. (2021). *What is a Buffer Overflow? How Do These Types of Attacks Work?* [online] SearchSecurity. Available at: https://www.techtarget.com/searchsecurity/definition/buffer-overflow.

Educative (N.D.). *What is the 'stack smashing detected' error?* [online] Available at: https://www.educative.io/answers/what-is-the-stack-smashing-detected-error [Accessed 26 Aug. 2023].

Welekwe, A. (2020). *Buffer overflow vulnerabilities and attacks explained*. [online] Comparitech. Available at: https://www.comparitech.com/blog/information-security/buffer-overflow-attacks-vulnerabilities/.

Tech Monger (N.D). *Producer Consumer Model in Python - Tech Monger*. [online] Available at: https://techmonger.github.io/55/producer-consumer-python/ [Accessed 30 Aug. 2023].

GeeksforGeeks. (2017). *Mutex lock for Linux Thread Synchronization - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/.

GeeksforGeeks. (2022). *Implement thread-safe queue in C++*. [online] Available at: https://www.geeksforgeeks.org/implement-thread-safe-queue-in-c/.

Python Tutorial (N.D.). *How to Use Python Threading Lock to Prevent Race Conditions*. [online] Available at: https://www.pythontutorial.net/python-concurrency/python-threading-lock/.