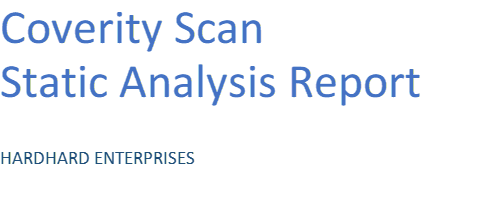
​



Statement of Intent

Overview

This document aims to provide a record of static code analysis performed on a specific issue from the Coverity SAST scan for the NASA ION Open-Source code 4.1.1 project.

The primary purpose of this document is to validate the issue identified via the automated detection process to eliminate false positives.

Depending on findings, secondary purposes can include but are not limited to listing/providing recommended fixes alongside a list of attack vectors and potential exploits for consideration.

Reporting Best Practices

Please ensure best practices are kept when completing the document via regularly updating the Acronyms and Abbreviations table alongside any iterations made to the Document History table. This will allow other members to identify any updates and progress made across trimesters easily.

When using code snippets, please use screenshots that are clear and easy to read, alternatively, use words built-in code formatter found [here](https://appsource.microsoft.com/en-us/product/office/WA104382008?tab=Overview).

Document Naming Conventions

Naming conventions for this file are as follow; SAR\_{CID}. For example, when investigating issue 123456 the file name would be SAR\_123456.docx

Document History

|  |  |  |  |
| --- | --- | --- | --- |
| **Dates** | **Version** | **Author** | **Comments** |
| 12/12/2022 | V0.1 | YIZHOU FENG/YONGXIANG HUANG | Initial document |
| 14/12/2022 | V0.2 | YIZHOU FENG/YONGXIANG HUANG | Update information and conclusions |
| 16/12/2022 | V0.3 | YIZHOU FENG/YONGXIANG HUANG | Final fix |

Table of Content

Contents

Introduction

Objective

The primary objective of this analysis is to determine whether the defects identified in the Coverity Report for the ION Open Source 4.1.1 project are:

* Indeed, defects.
* Potentially exploitable.

The secondary objective of this analysis, where applicable, is to provide the following:

* Recommendation(s) to fix.
* Any exploit for consideration.

Scope

This static code analysis is limited to the ***Resource leak*** type defect identified in the following CIDs: 1520844

Acronyms and Abbreviations

*Please keep an updated list of acronyms and abbreviations used throughout the report.*

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| DTN | Delay/Disruption Tolerant Network |
| ION | Interplanetary Overlay Network |
| CID | Coverity Issue Identification Number |

Code Review and Analysis

Outcomes

When performing static code analysis using the ION Open Source 4.1.1 dashboard for CID 1520844, this vulnerability is flagged as a high severity vulnerability and we believe the presence of this vulnerability poses a high risk to all code. Also, the presence of the return\_constant flaw and the overrun-buffer-arg.

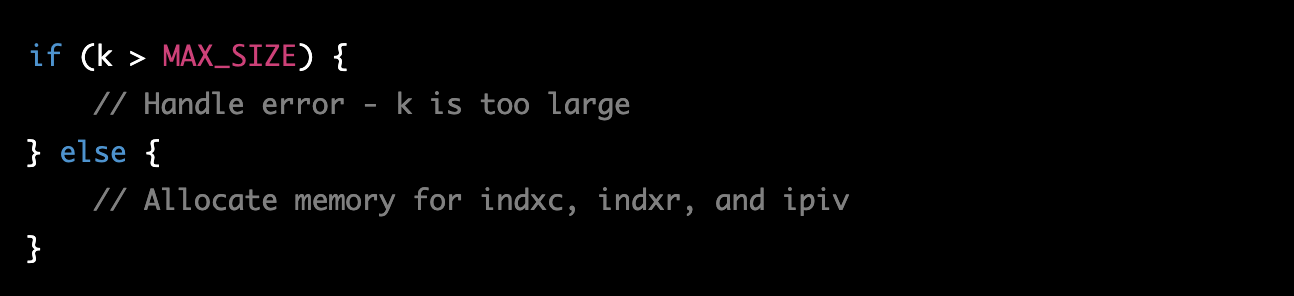
Observations

*For* **alloc\_fn:** Storage is returned from allocation function malloc.

The error message **alloc\_fn: Storage is returned from allocation function malloc** indicates that the **malloc** function, which is used to allocate memory dynamically at runtime, has returned a null pointer. This typically indicates that there is not enough memory available to allocate the requested amount of memory.

In the code snippet you provided, the **malloc** function is being used on lines 238, 239, and 240 to allocate memory for three arrays: **indxc**, **indxr**, and **ipiv**. The size of each array is determined by the **k** parameter, which is passed to the **232\_invert\_mat** function when it is called.

One possible solution to this error is to check the value of the **k** parameter before calling the **malloc** function, and make sure that it is not set to a value that is too large. For example, you could add a conditional statement like this:



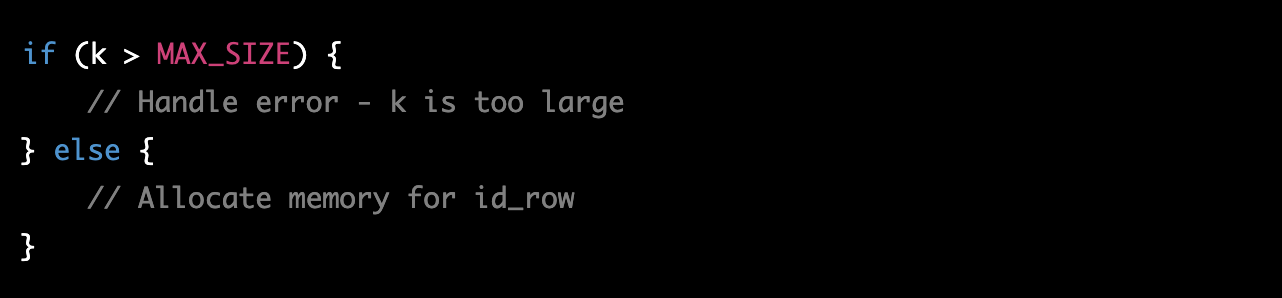
Here, **MAX\_SIZE** is a constant that you can define to specify the maximum size that the **k** parameter is allowed to be. This will prevent the **malloc** function from being called with a value of **k** that is too large, and should fix the **alloc\_fn** error.

For **var\_assign:** Assigning: id\_row = storage returned from malloc(1U \* k).

The error message **var\_assign: Assigning: id\_row = storage returned from malloc(1U \* k)** indicates that the **malloc** function, which is used to allocate memory dynamically at runtime, has returned a null pointer. This typically indicates that there is not enough memory available to allocate the requested amount of memory.

In the code snippet you provided, the **malloc** function is being called on line 238 to allocate memory for the **id\_row** array. The size of the array is determined by the **k** parameter, which is passed to the **232\_invert\_mat** function when it is called.

One possible solution to this error is to check the value of the **k** parameter before calling the **malloc** function, and make sure that it is not set to a value that is too large. For example, you could add a conditional statement like this:



Here, **MAX\_SIZE** is a constant that you can define to specify the maximum size that the **k** parameter is allowed to be. This will prevent the **malloc** function from being called with a value of **k** that is too large, and should fix the **var\_assign** error.

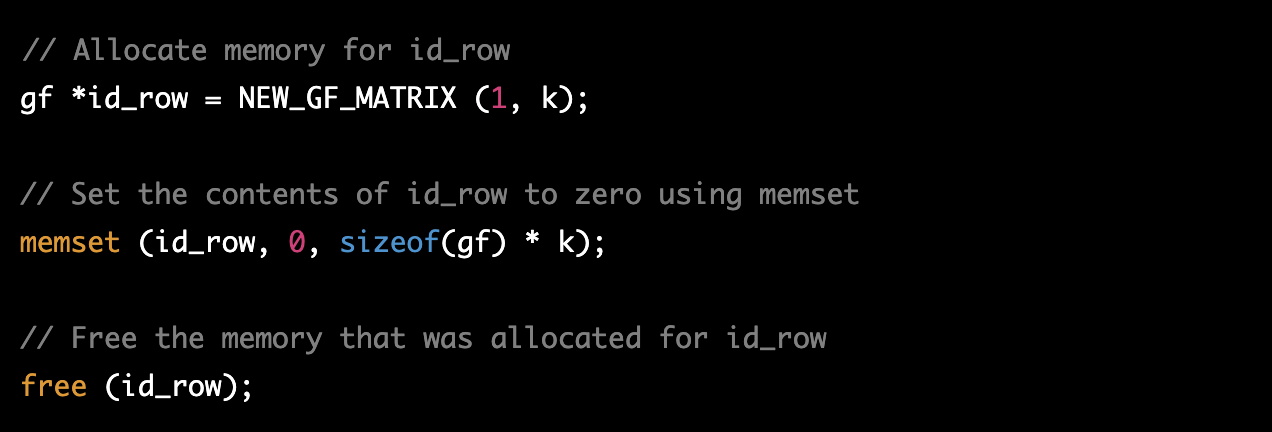
For **noescape:** Resource id\_row is not freed or pointed-to in memset.

The error message **noescape: Resource id\_row is not freed or pointed-to in memset** indicates that a memory leak has occurred in your code. A memory leak occurs when a program allocates memory dynamically, but does not release it properly when it is no longer needed. This can cause your program to run out of memory and crash, or to behave unpredictably.

In the code snippet you provided, the **id\_row** array is allocated using the **NEW\_GF\_MATRIX** function on line 241. Later, the **memset** function is called on line 244 to set the contents of this array to zero. However, the error message indicates that the **id\_row** array is not freed or pointed to in the **memset** function, which means that it will not be released properly when it is no longer needed.

To fix this error, you will need to make sure that the **id\_row** array is properly freed or pointed to in the **memset** function. This could involve calling the **free** function to release the memory that was allocated for the **id\_row** array, or assigning the **id\_row** array to a pointer variable so that it can be accessed and freed later.

For example, you could modify the code like this:



This should fix the **noescape** error and prevent the memory leak in your code. Let me know if you have any other questions or if you need any further clarification on this.

For **leaked\_storage:** Variable id\_row going out of scope leaks the storage it points to.

The error message **leaked\_storage: Variable id\_row going out of scope leaks the storage it points to** indicates that a memory leak has occurred in your code. A memory leak occurs when a program allocates memory dynamically, but does not release it properly when it is no longer needed. This can cause your program to run out of memory and crash, or to behave unpredictably.

In the code snippet you provided, the **id\_row** array is allocated using the **NEW\_GF\_MATRIX** function on line 241. However, the error message indicates that the **id\_row** array is not properly freed or released when it goes out of scope. This means that the memory that was allocated for the **id\_row** array will not be released, and will remain allocated until the program terminates.

To fix this error, you will need to make sure that the **id\_row** array is properly freed or released when it goes out of scope. This can be done by calling the **free** function to release the memory that was allocated for the **id\_row** array.

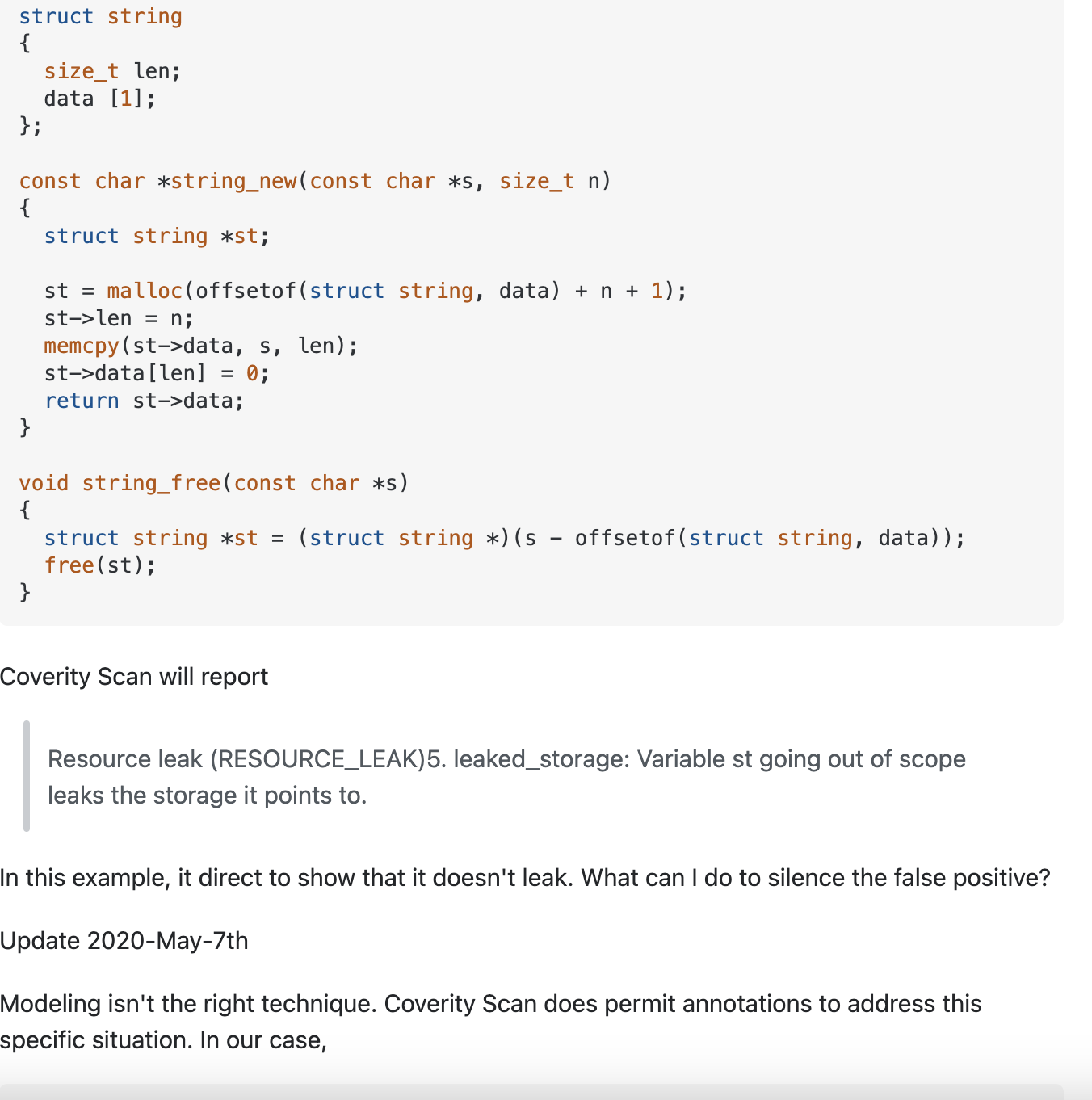
For example, we could modify the code like this:

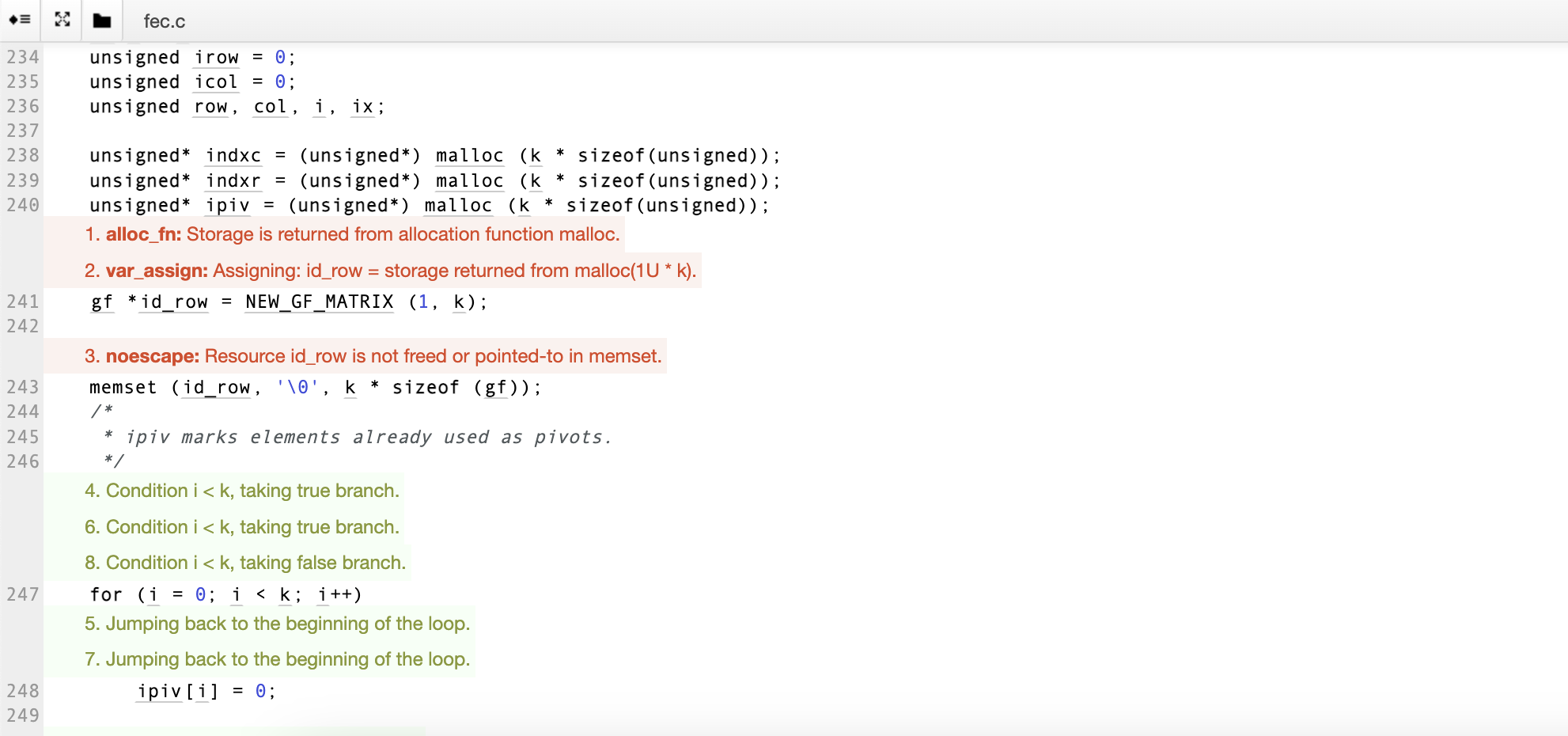
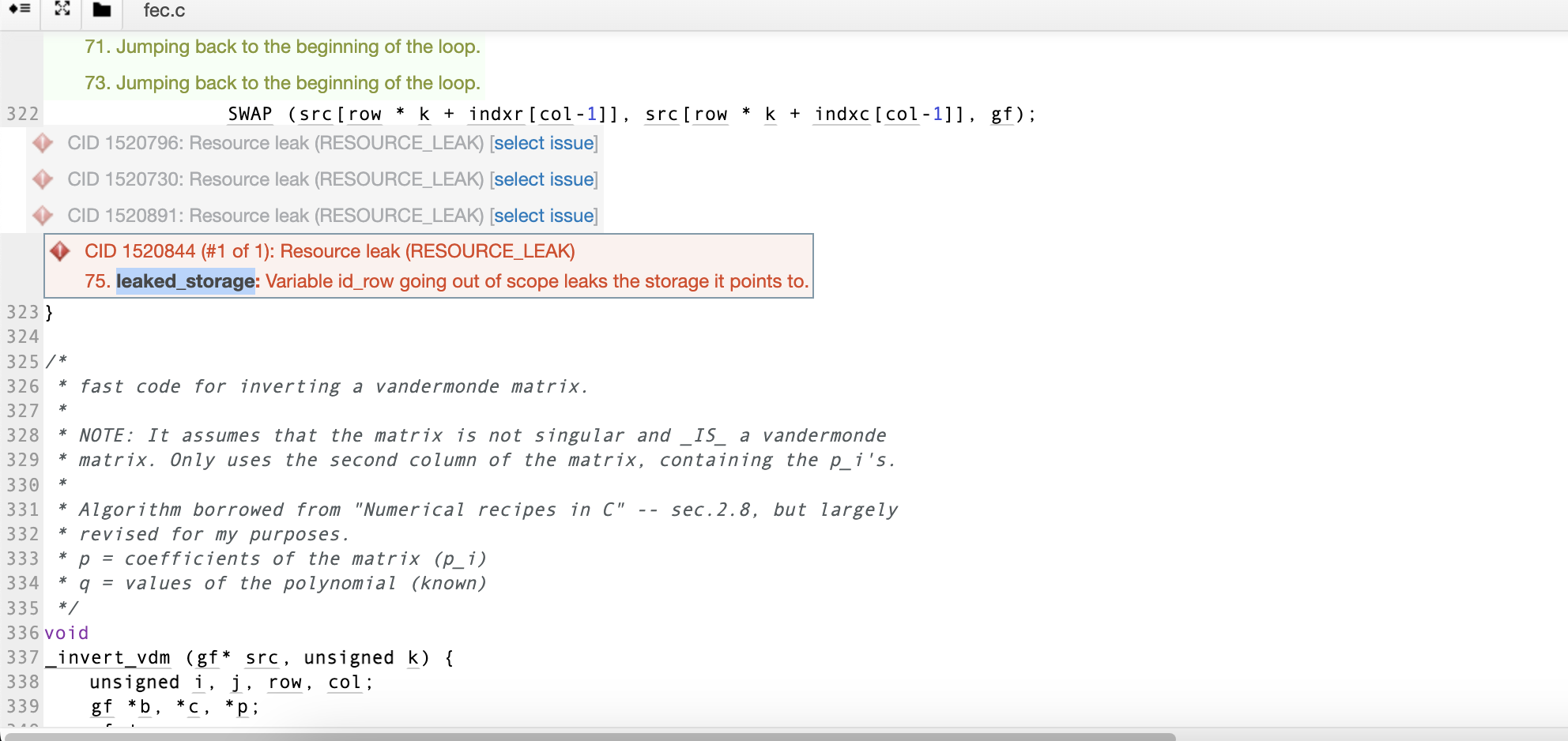
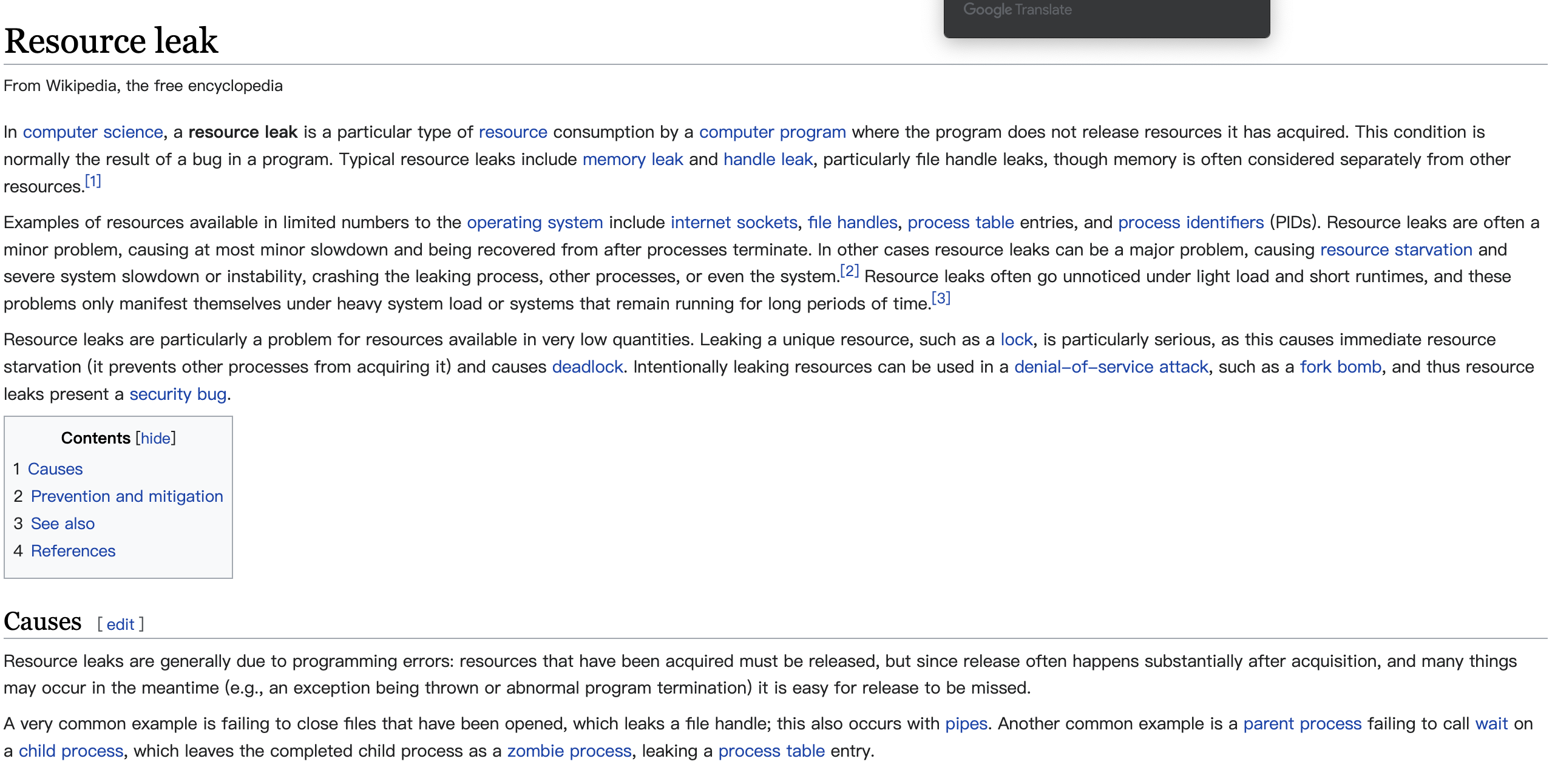
文本

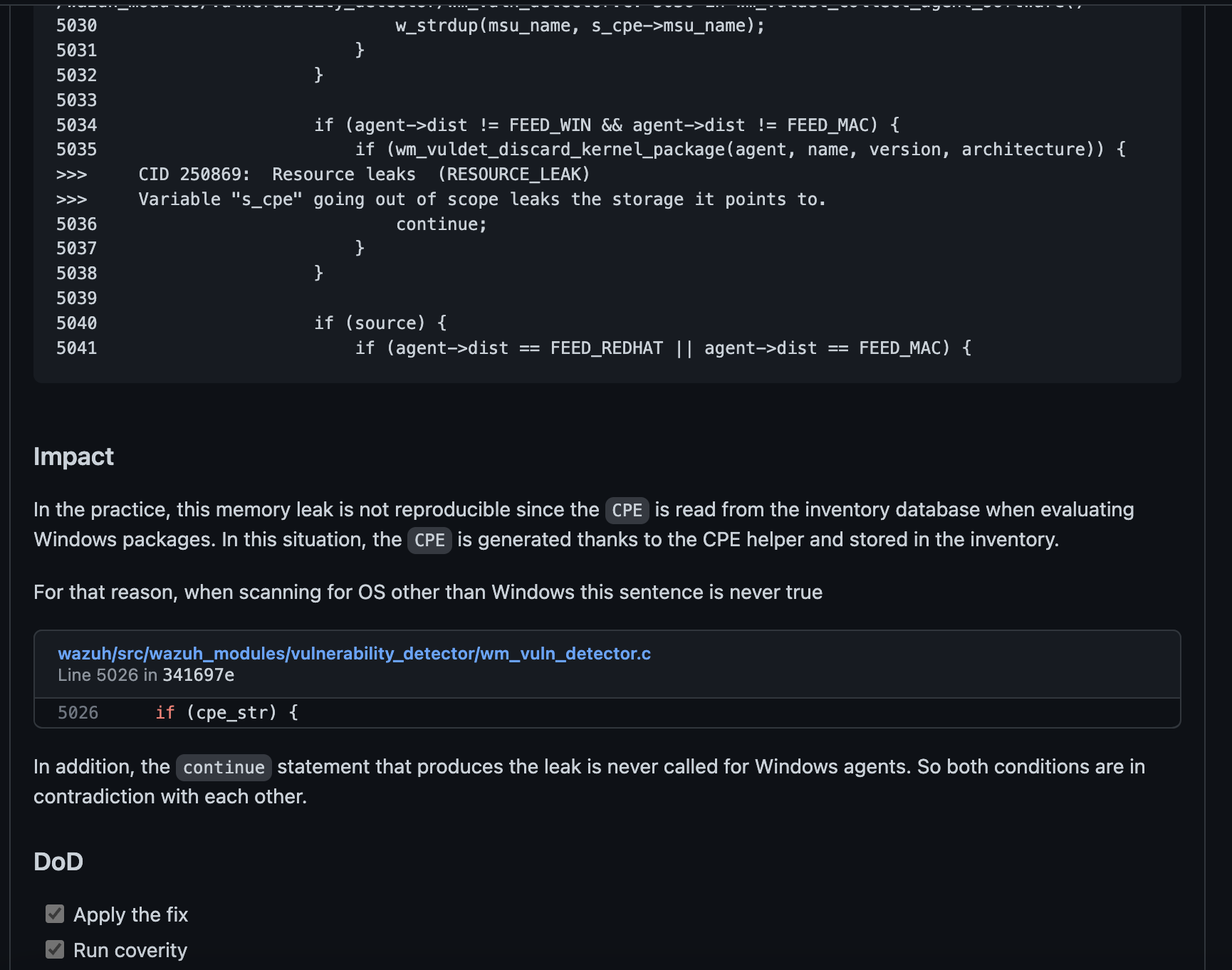
描述已自动生成

This should fix the **leaked\_storage** error and prevent the memory leak in your code. Let me know if you have any other questions or if you need any further clarification on this.

Supporting Evidence





Conclusions and Recommendations

*In general, the code is a C implementation of a forward error correction (FEC) library, specifically the Reed-Solomon code. FEC is a technique used to detect and correct errors that may occur in data transmission or storage. The Reed-Solomon code is a specific type of FEC code that can be used to correct errors in data transmission or storage.*

*The code you provided is written in C, a popular programming language that is commonly used for systems-level programming and low-level application development. The code contains various functions and data structures that are used to implement the Reed-Solomon code, as well as some utility functions for performing common operations such as memory allocation and array manipulation.*

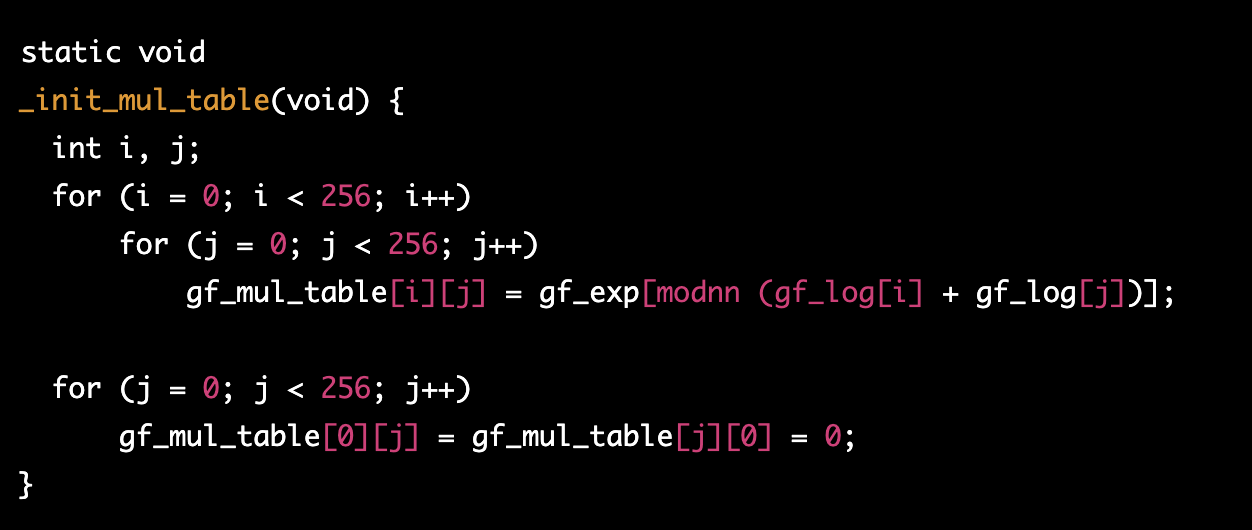
*One of the key data structures used in the code is the gf type, which represents a finite field element. A finite field, also known as a Galois field, is a mathematical structure that is used in the implementation of the Reed-Solomon code. A finite field is a set of numbers that can be added, subtracted, multiplied, and divided in a well-defined manner, subject to some constraints. In the code you provided, the gf type is used to represent elements of a finite field of size 2^8 - 1 (i.e. a field with 256 elements).*

*The \_init\_mul\_table function is used to initialize the multiplication table for the finite field. This table is used by the gf\_mul macro, which is defined on line 60, to perform efficient multiplication operations on finite field elements. The multiplication table is* into what is causing the error.

Populated with values from the gf\_exp and gf\_log arrays, which contain pre-computed logarithm and exponent values for the finite field.

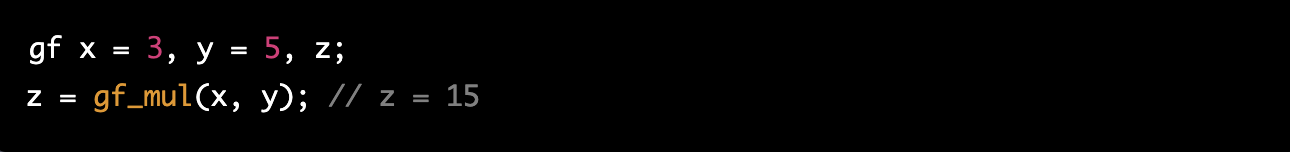
The \_init\_mul\_table function is called only once, when the FEC library is initialized. After the multiplication table has been initialized, the gf\_mul macro can be used to efficiently perform multiplication operations on finite field elements.

Here is the complete implementation of the \_init\_mul\_table function:



This function initializes the gf\_mul\_table array with values that can be used to efficiently perform multiplication operations on finite field elements. It does this by looping through all possible pairs of finite field elements, and using the gf\_exp and gf\_log arrays to compute the result of the multiplication and store it in the multiplication table.

Once the multiplication table has been initialized, the gf\_mul macro can be used to perform multiplication operations on finite field elements. For example, the following code uses the gf\_mul macro to multiply two finite field elements x and y and store the result in the z variable:

 The gf\_mul macro uses the pre-computed values in the gf\_mul\_table array to quickly and efficiently perform the multiplication operation. This allows the Reed-Solomon code to perform multiplication operations on finite field elements without having to compute them from scratch every time.

I hope this helps. Let me know if you have any other questions or if you need any further clarification on this.

References   
*Please keep an updated references list in APA7; The Deakin referencing guide can be found* [here](https://www.deakin.edu.au/__data/assets/pdf_file/0009/2236752/Deakin-guide-to-APA7.pdf)*.* 

Appendix

*Include additional information/documentation here to help the readers understand complex information.*