Coverity Scan Static Analysis Report

Hardhard Enterprises

**T3** 2022

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 formatted 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** |
| 29/11/2022 | 1.0 | Jesse Ludeman | Initial document |
| 30/11/2022 | 1.1 | Jesse Ludeman | Update conclusions and recommendations |
| 1/12/2022 | 1.2 | Jesse Ludeman | Add call graph and add origin of tainted data |

Table of Content

Contents

[Introduction 3](#_Toc119848724)

[Objective 3](#_Toc119848725)

[Scope 3](#_Toc119848726)

[Acronyms and Abbreviations 3](#_Toc119848727)

[Code Review and Analysis 4](#_Toc119848728)

[Outcomes 4](#_Toc119848729)

[Observations 4](#_Toc119848730)

[Supporting Evidence 4](#_Toc119848731)

[Conclusions and Recommendations 4](#_Toc119848732)

[References 4](#_Toc119848733)

[Appendix 4](#_Toc119848734)

# 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 ***Insecure data handling*** type defect identified in the following CIDs: 1520892

# 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 |
|  |  |

# Code Review and Analysis

## Outcomes

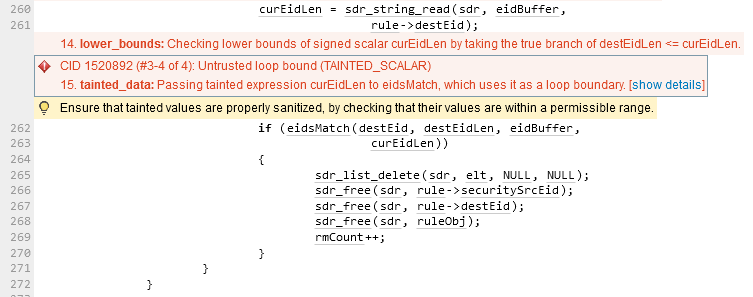
When performing static code analysis using the ION Open Source 4.1.1 dashboard for CID 1520892, there is a medium impact problem that involves an untrusted loop bound executing inside a procedure. Additionally, there is also a function that returns tainted data.

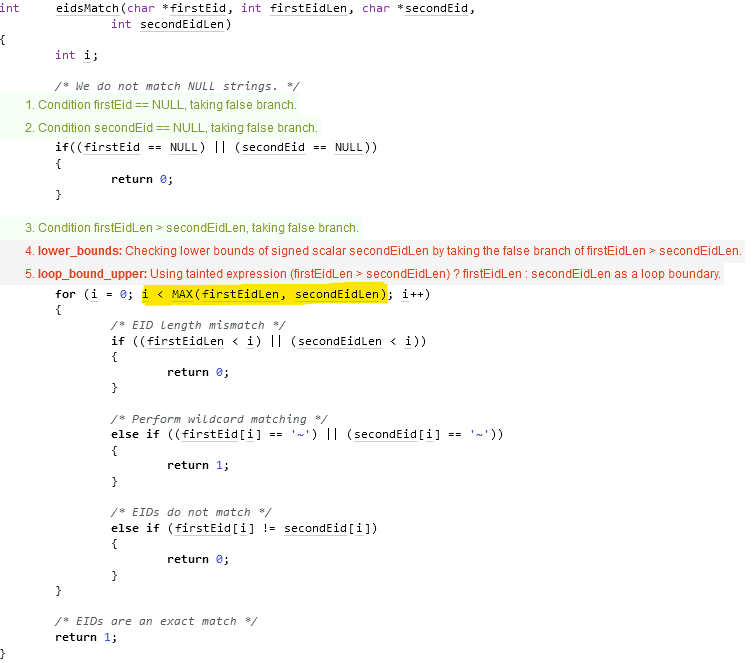
## Observations

An “untrusted loop bound” type error indicates a loop operation has been called, but there exists no condition that will stop it effectively. This results in an excessive loop that could lead to an undesirable outcome or other consequences.

The code snippet below demonstrates where the offending issue lies. Note that the expression curEidLen is passed to eidsMatch function in the if statement.

## Supporting Evidence

  
  
Figure 1 – Initial call to eidsMatch function

  
Figure 2 – eidsMatch function

Note in the highlighted code in the above for loop, that the i loop variable is compared against the MAX value which is then passed firstEidLen and secondEidLen values. Both of these are int data types.

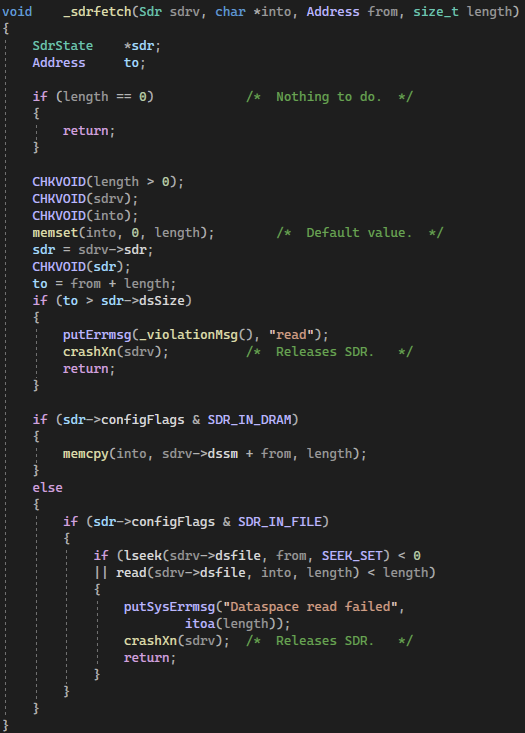
  
Figure 3 – MAX variable

The MAX macro above is using a conditional expression. This is comparing the A and B values, and if the condition is true; then it evaluates to A, otherwise to B. The problem with this expression is that there’s currently no way to validate what’s being passed to these variables.

Furthermore, in order to identify where the offending tainted data is originating from, Hardhat Enterprises has generated a call graph.

Diagram

Description automatically generated  
Figure 4 – Call graph

  
Figure 5 – \_sdr\_fetch procedure

Note that in the \_sdr\_fetch procedure, that the length variable is passed in and validated. If this value evaluates to 0, then it returns. Therefore, the origin of the tainted data happens to be a false positive.

# Conclusions and Recommendations

A tainted value is being used as a loop bound can be a serious problem, and can be exploited by an attacker. The value is considered to be tainted when it enters the program via an input operation elsewhere. If the loop value has in fact been tainted, and is being used as a loop boundary, then this can cause the loop to iterate a very large number of times. This can subsequently cause the program to stop execution, and therefore introduces a denial of service attack by stopping the program and consuming a large amount of time and computing resources.

The suggestion here would be to update or replace the MAX function and add functionality that validates the input variables for A and B. For example, this expression does not check and validate if there’s an upper and lower bound being passed to the macro. Furthermore, implementing this would ensure that excessively large or smaller integers would not cause an overflow or other type range problem during program execution.

|  |
| --- |
| /\* Define the upper and lower bound values\*/  #define MAX\_A 500  #define MAX\_B 500  #include <stdio.h>  int MAX(int *a*, int *b*)  {      /\* Ensure that both input values do not exceed their sizes in memory \*/      if (*a* > MAX\_A || *b* > MAX\_B)      {          printf("Input values are too high!\n");          return 0;      }      else if (*a* > *b*)      {          return *a*;      }      else      {          return *b*;      }  } |

Figure 6 – Example code with updated MAX function

The above example demonstrates how to validate the input for both A and B integer values. Note that if both values exceed a given upper and lower bound, the caller will be advised and the function will fail gracefully. This is an appropriate way to handle and check for a tainted value being used as a loop boundary.

References

Appendix

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