D4.3

Benchmarks for evaluating VESSEDIA tools

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D4.3 /1.0
Work package contributing to the deliverable:	WP 4
Due date:	2018-10-31
Actual submission date:	29th of October 2018

Responsible organisation:	AMO
Editor:	Cédric BERTHION
Dissemination level:	PUBLIC
Revision:	V1.0

Abstract:	This report will present benchmarks for evaluating the quality of VESSEDIA tools regarding both simple and complex security vulnerabilities.
Keywords:	Security evaluation C Source code Code auditing Frama-C

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Cédric BERTHION (AMO)

Contributors (ordered according to beneficiary numbers)

Mounir KELLIL (CEA)

Virgile PREVOSTO (CEA)

Nicolas DATIN (AMO)

Executive Summary

This document presents benchmarks for evaluating the quality of VESSEDIA tools regarding vulnerability detection in a security evaluation context. This document focuses on security audit of C source code with specific constraints faced by security evaluators: time constraints and lack of previous knowledge of the audited source code.

Samples in theses benchmarks were selected on the following criteria. They implement well-known C vulnerabilities, such as Buffer overflow, Use-After-Free… Source code should be legacy code with no additional specification (Frama-C ACSL…). The goal is to qualify the effectiveness of the tools without requiring any heavy work from the evaluator. Minimum specifications should be added for each test case. Finally, the benchmarks should present different levels of complexity for the analysis, from very simple piece of code to samples based on real-life software.

Each test case is detailed (cf. [D43.zip]). The vulnerability it implements is presented and the reasons why this sample was chosen are explained. Moreover, an expected result for the analysis tools is given to make it easy to check if the vulnerability is detected or not.

Besides, this document presents other static analysis tools for auditing C source code in order to compare them to VESSEDIA tools. A methodology with metrics is presented to explain how the tests will be run.

Based on these benchmarks, document [D4.5] will test the quality and the limits of VESSEDIA tools.

Contents

List of Tables

Chapter 1 Introduction

1.1 VESSEDIA motivation and background

The VESSEDIA project aims to bring safety and security to the next generation of software applications and the Internet-connected devices. In our rapidly changing world, the Internet has been the source of many benefits for individuals and companies alike, transforming entire industries. With this new technology, capable of connecting billions of devices and people together, new threats have also appeared – threats VESSEDIA will help software developers address in order to create connected applications that are safe and secure. VESSEDIA proposes to enhance and scale up modern software analysis tools, in particular the mostly used open-source Frama-C analysis platform, to make them useful and accessible to a wider audience of developers of connected applications. At the forefront of connected applications is the Internet of Things (or IoT for short), which has undergone explosive growth and where security risks have become all too real. VESSEDIA will focus on this domain to demonstrate the benefits our tools bring to the table when developing connected applications. VESSEDIA will tackle this challenge by 1) developing a methodology that makes it possible to adopt and use source code analysis tools as efficiently and with similar benefits as it is already possible in the case of highly critical applications, 2) enhancing the Frama-C toolbox to enable efficient and fast implementation, 3) demonstrating the capabilities of the new toolbox on typical IoT applications, including an IoT Operating System (Contiki), 4) developing a standardisation plan for generalising the use of the toolbox, 5) contributing to the Common Criteria certification process, and 6) defining a “Verified in Europe” label for validating software products with European technologies such as Frama-C.

1.2 Role of the Deliverable

This document presents benchmarks for evaluating the quality of VESSEDIA tools regarding security vulnerabilities detection in a security evaluation context. It focuses on security audit of C source code with specific constraints faced by security evaluators: time constraints and lacks of previous knowledge of the audited source code. Each test case is detailed. Moreover, it presents the methodology and the metrics used to test the quality and the limits of VESSEDIA tools.

Based on this document, document [D4.5] will present the results of this benchmarking.

1.3 Structure of the document

The document can be divided into 3 major parts:

Chapter 2 describes the methodology of the benchmarking. It presents the context of code source analysis in security evaluations. Based on that context, the choice of the samples is explained. Then, Chapter 3 specifies the test environment of the benchmarks. Finally, individual test cases are detailed in Chapter 4.

1.4 Related Deliverables

Deliverable Number	Deliverable Title	Type	Dissemination level
[D1.7]	Vulnerability discovery methodology	Report	Public
[D4.2]	VESSEDIA approach for security evaluation	Report	Public
[D43.zip]	Source code of samples selected for the benchmarks	Archive	Public
[D4.5]	Quality tests & limits of VESSEDIA tools regarding security vulnerabilities detection	Report	Public

Table 1: Related Deliverables

The document [D1.7] presents a methodology for discovering vulnerabilities with Frama-C. Code samples used in this document were added to the benchmarks.

The document [D.4.2] describes a proposed security evaluation methodology for VESSEDIA project. In particular, this document describes VESSEDIA tools.

The document [D4.5] is due to M36 (December 2019) will present the results of these benchmarks.

Source code of samples detailed in this document are available in [D43.zip].

Chapter 2 Methodology

This chapter describes the methodology of the benchmarking. It presents the context of code source analysis in security evaluations.

2.1 Evaluator / Developer Constraints

Source code analysis is used by at least two different kinds of users during the life-cycle of a security-sensitive product: developers and security evaluators. Source code analysis is used for testing and bug tracking by developers. A good coding behaviour is to integrate some source code analysis tools to the project. Once an analysis tool is set up, it can be used all along the project on each version of the code. Developers have full knowledge of their product so they can finely tune the analyser to avoid false positives. Remaining false positives can be checked and documented in order to make the next analysis always more precise. Thus, source code analysis is also part of source code maintenance in the life-cycle of a product.

In another hand, security evaluators use source code analysis to find vulnerabilities in a product. They are using source code analysis tools like “vulnerability scanners”. They do not have a full knowledge of the product. Depending of the documentation that has been provided by the developers, reverse engineering has to be performed to understand the role of each module of the project. Thus, evaluators cannot check the whole code of a program because it would be too time consuming. They have to focus on critical modules where critical vulnerabilities are more likely to be. In the same way, analysers’ outputs have to be as synthetic as possible. The goal of security evaluators is not to discover every vulnerability in the product. Detecting one critical vulnerability may be enough to set a verdict for the evaluation and to consider the product as non-secure.

To sum up, developers need source code analysers to avoid as many bugs as possible, whereas security evaluators need automatic tools to quickly find some vulnerabilities even if they are missing some of them.

Regarding VESSEDIA tools, Frama-C is the one that is more likely to be used for code audit in a security evaluation context. Indeed, the plugin EVA enables evaluators to use it as a “vulnerability scanner”. One of the main constraints for security evaluators is the duration of evaluations. They may only have a week or two to analyse a product. Most products do not use of formal code analyser. Hence, evaluators must be able to deal with them without having to finely annotate the code.

The C code samples of the benchmarks are not annotated and their analyses should be performed as automatically as possible with as few manual interventions as possible. That is why these benchmarks focus on Frama-C and its plugin EVA. This plugin is also the easiest to use on non-annotated code.

2.2 Benchmarks for vulnerability detection

Three projects were studied to provide samples for these benchmarks: the Juliet[1] Test Suite from NIST, SV-COMP[2] from TACAS and the DARPA Cyber Grand Challenge[3].

The Juliet Test Suite is a collection of test cases in the C/C++ language. It contains examples organised under different CWEs. This test suite was created by the National Security Agency’s (NSA) Center for Assured Software (CAS) and developed specifically for assessing the capabilities of static analysis tools. The test suite is well organised and well documented. Each test case is

[1] https://samate.nist.gov/SRD/testsuite.php
[2] https://sv-comp/sosy-lab.org/2019
[3] https://www.darpa.mil/program/cyber-grand-challenge

described with its CWE category (for instance CWE 415 for Double Free) and the potential flaws in the source code are well commented.

After a review of the proposed samples, it appears that the use cases are quite too simple and not well adapted to our purpose. The test suite is composed of minimalistic test cases of one hundred lines of code. These sample are neither complex enough nor very realistic. One of the main challenges for static analyser is the ability to perform analysis on “real world” projects. Note that this critic is confirmed by the user guide “Juliet Test Suite v1.2 for C-Cpp – User Guide.pdf”[4] in the chapter §1.3.2 “Limitations of the Tests Cases”. It is explained that the test cases are simpler than natural code and that analysis tools may report flaws in the test cases that they would rarely report in natural, non-trivial code.

SV-COMP is a competition of software-verification tools. Its goal is to widely distribute a benchmark suite of verification tasks in order to compare state-of-the-art verification tools. The C samples are available at https://github.com/sosy-lab/sv-benchmarks. Some test cases implement security flaws but most are verification oriented and do not deal specifically with security issues.

The third benchmark suite that was studied is the test suite of the DARPA Cyber Grand Challenge (CGC). DARPA explains that today, the process of finding and countering bugs, hack and other security infection vectors is still effectively artisanal. Thus, the CGC is a competition to create an automatic defensive system capable of finding flaws and to patch them in real time. Each team proposed an automatic system able to analyse binaries for twelve hours. The source code of these binaries is available on the following GitHub repository: https://github.com/trailofbits/cb-multios. Each of the three hundred C/C++ samples are documented. The implemented vulnerabilities are explained and identified by their CWE number. The test cases are quite complex. They are made of several source files and more than one thousand lines of code. They can be considered as “realistic” in the sense that they have a real purpose (simulating a movie rental system for instance), that is to say, it’s not only a vulnerability demonstration. Moreover, they do not use too many dependencies which will help for analysing without adding stubs for external libraries.

Thus, the DARPA CGC present non-trivial, well documented and security oriented code samples. This is why these benchmarks are based on samples from it.

2.3 Building a Corpus

Our goal is to use Frama-C like a “vulnerability scanner” with as few manual interactions as possible. However, some annotation may still be required, and/or some stubs, describing external library, may need to be implemented. Thus, these benchmarks cannot just be the whole set of the three thousand samples of the CGC.

The challenges were chosen to focus on eight of the most well-known vulnerabilities in C which are:

- Stack Buffer Overflow (CWE-121);
- Heap Buffer Overflow (CWE-122);
- Null Pointer Dereference (CWE-476);
- Off By One (CWE-193);
- Use After Free (CWE-416);
- Uninitialised Variable (CWE-457);
- Double Free (CWE-415);
- Format String (CWE-134).

An example for each of them and a short description of the way to exploit them is also presented in this section.

For each of these well-known vulnerabilities, two examples were chosen for these benchmarks. One is a complex sample from the CGC and the second one is a toy example. The goal of including the

[4] https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf

toy example is to check that the analysed tool is able to detect the corresponding vulnerability. Samples from the CGC are bigger and present a tougher challenge to the analyser.

To sum up, the final corpus is composed of 16 samples that illustrated 8 different vulnerabilities. For each vulnerability, a toy sample will allow us to check the bare capability of an analyser and a more complex sample will allow to check the capabilities of the tool to handle "real-world software".

2.3.1 Vulnerability 1 - Stack Buffer Overflow

```c
#include <string.h>

void fill(char *buf, char *arg) {
    for (int i = 0; i < strlen(arg); i++) {
        buf[i] = arg[i];
    }
}

int main(int argc, char **argv) {
    char name[16] = {0};

    if (argc < 2)
        return 1;
    fill(name, argv[1]);
    return 0;
}
```

In the above example, an array of type char is declared with a size of 16 elements. When *fill* is called, all bytes from *arg* will be copied to *buf* and then create a buffer overflow condition.

This vulnerability allows an attacker to overwrite data on the stack. It can lead to an arbitrary code execution in the vulnerable process context if the return address of the preceding function's stack frame is impacted. This example represents one of the most basic form of this vulnerability (fixed buffer size and buffer with unchecked boundaries).

Static analysers should be able to detect this kind of vulnerability in its simplest form. Ideally, the analyser should assert that a destination buffer is large enough to contain the source data.

2.3.2 Vulnerability 2 – Heap Buffer Overflow

```c
#include <string.h>
#include <stdlib.h>

void fill(char *buf, char *arg) {
    for (int i = 0; i < strlen(arg); i++) {
        buf[i] = arg[i];
    }
}

int main(int argc, char **argv) {
    char *name;

    if (argc < 2)
        return 1;
    name = malloc(16);
    fill(name, argv[1]);
    return 0;
}
```

This example illustrates a heap buffer overflow vulnerability caused by the copy of a buffer with unchecked boundaries in an allocated space on the heap.

A heap buffer overflow is a type of buffer overflow that occurs in the heap data area. Heap area is used to store dynamically allocated data at runtime and generally contains program data. Exploitation is performed by corrupting data on the heap in specific ways to overwrite internal structures such as linked list pointers used to manage metadata of allocated chunks, depending on the internal memory manager used by the system. This could lead to a memory leak or code execution.

A static analyser should detect overflows by comparing number of written bytes with the initial allocation size.

2.3.3 Vulnerability 3 – Null pointer dereference

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  
4  int *input_num(int *a) {
5      printf("Enter a non-negative number: ");
6      scanf("%d", a);
7      if (*a < 1) {
8          a = 0;
9      }
10     return a;
11 }
12 
13 int main(int argc, char **argv) {
14     int *num1;
15     int *num2;
16     int result;
17 
18     num1 = malloc(sizeof(int) * 1);
19       if (num1 == NULL) {
20             return 1;
21       }
22     num2 = malloc(sizeof(int) * 1);
23       if (num2 == NULL) {
24             return 1;
25       }
26 
27     num1 = input_num(num1);
28     num2 = input_num(num2);
29     result = *num1 + *num2;
30     printf("result: %d\n", result);
31 
32     return 0;
33 }
```

A null-pointer dereference takes place when a pointer with *NULL* as value is used as if it pointed to a valid memory area. This vulnerability usually results in a denial of service because of process crash, or, if the exception is handled by the process itself, a partial control of the execution flow.

This example is a basic calculator doing only a sum of two positive numbers. The problem is in the *input_num* function. It takes an integer pointer and returns it after it was filled by *scanf* asking the user to input a positive number. If the provided number is lower than 1, the conditional statement is executed (line 8). Instead of setting the value pointed by the pointer to 0, the pointer value itself is set to 0, resulting in a crash when the pointer is dereferenced on line 29.

Static analysers should warn if any pointer having an address inferior to 4096 is dereferenced. This value is generally specified by *mmap_min_addr* mitigation mechanism which prevent from mapping memory below that value.

2.3.4 Vulnerability 4 – Use after free

```
1  #include <malloc.h>
2  #include <stdio.h>
3  
4  typedef struct example {
5        void (*vulnfunc)();
6  } example;
7  
8  void good(){
9        printf("GOOD\n");
10 }
11 
12 void bad(){
13       printf("BAD\n");
14 }
15 
16 void helper_call_goodfunc(example *uafme){
17       example *private_uafme = uafme;
18       private_uafme->vulnfunc = good;
19       private_uafme->vulnfunc();
20       free(private_uafme);
21 }
22 
23 int main(int argc, const char * argv[]){
24       example *malloc1 = malloc(sizeof(example));
25       helper_call_goodfunc(malloc1);
26 
27       long *malloc2 = malloc(0);
28       *malloc2 = (long)bad;
29 
30       malloc1->vulnfunc();
31       return 0;
32 }
```

Use-after-free vulnerability is caused by the use of an already freed dangling pointer. In fact, this vulnerability takes advantage of the internal mechanism of the memory manager (for example *ptmalloc2*[5]). Indeed, when a pointer is freed, value is not zeroed and *ptmalloc* doesn't warn when a chunk that has been freed is used. By allocating a chunk of the same size of a previously freed one, an attacker can control its content and, if the chunk is used later, execute arbitrary code.

In this example, the *helper_call_good_func* function assigns a value to the function pointer before calling it and finally freeing it. However, when the *malloc2* variable is allocated at line 27, the memory manager will place it at the same address as previously allocated *malloc1.* The "malloc(0)" instruction is intended. In fact, the requested size of the allocation doesn't have to be equal to the previous one in the source code, the memory manager will likely place it in the previous block for performance reasons (is the new size is not greater). So, by modifying content of *malloc2*, an attacker can control execution flow because of the re-use of the *malloc1* variable at line 30.

Static analysers should save the state of any pointers (allocated, freed) and throw warnings if a freed pointer is used.

[5] Multi-threaded memory manager: https://github.com/emeryberger/Malloc-Implementations/tree/master/allocators/ptmalloc/ptmalloc2

2.3.5 Vulnerability 5 – Off by one

```
1   #include <string.h>
2   #include <stdlib.h>
3
4   #define NAME_MAX 16
5
6   void fill(char *buf, char *arg) {
7       for (int i = 0; arg[i] != 0, i <= NAME_MAX; i++) {
8           buf[i] = arg[i];
9       }
10  }
11
12  int main(int argc, char **argv) {
13      char name[NAME_MAX];
14
15      if (argc < 2)
16          return 1;
17      fill(name, argv[1]);
18      printf("Your name is %s\n", name);
19      return 0;
20  }
```

Off-by-one bugs are very common but often hard to detect and exploit. In this example, the bug comes from an improper stop condition at line 7. The “less than or equal” conditional statement will create an overflow of one byte, overwriting stack data and leading to arbitrary read condition. By overwriting least significant byte of the *rbp* register, an attacker could also control the execution flow by jumping below in the stack layout and execute arbitrary code.

As for a classical stack buffer overflow, static analysers should throw an out of bounds write error if any byte is written outside the buffer’s limits.

2.3.6 Vulnerability 6 – Uninitialised variable

```
1   #include <stdio.h>
2
3   void number_args(int i) {
4       printf("Number of arguments: %d\n", i);
5   }
6
7   void access_user_panel() {
8       puts("Welcome to user panel.");
9   }
10
11  void access_admin_panel() {
12      puts("Welcome to admin panel.");
13  }
14
15  void panel(void) {
16      int is_admin;
17
18      switch (is_admin) {
19          case 1:
20              access_user_panel();
21              break;
22          case 2:
23              access_admin_panel();
24              break;
25          default:
```

```
26                  access_user_panel();
27          }
28  }
29
30  int main(int argc, char **argv) {
31          number_args(argc);
32          panel();
33          return 0;
34  }
```

Uninitialised variable based vulnerability happens when a variable is declared but not initialised. In this case, the system will automatically set a value to the newly created variable. However, this value is not random nor null, it will be chosen from a near region. By controlling this region an attacker can therefore control the value of the new variable and alter the logic of the program or run arbitrary code.

This example illustrates this vulnerability with an uninitialised variable used to check if a user could access simple user or admin panel. With the call to *number_args* function, the parameter "argc" will be spread on the stack. Later, on line 16, the *is_admin* variable is declared uninitialised and might then take the *argc* value because it lands in the same region. If a user gives an argument to this program, the *argc* and so *is_admin* variables will have 2 as value and let the user access the admin panel.

A static analysis should reveal all uninitialised variables that are used, even if the potential initialization is executed only in a conditional block.

2.3.7 Vulnerability 7 – Double free

```
#include <stdlib.h>

int main(void)
{
        char *a, *b;
        a = malloc(64);
        free(a);
        b = malloc(64);
        free(a);

        return 0;
}
```

A double free vulnerability occurs when a variable is freed twice. The *malloc* memory manager sorts chunks by their size and, for performance reasons, do not cleanup content when freed. So, when a chunk is freed and after that a same-sized chunk is newly allocated, *malloc* will place it at the same address in the heap.

In this example, when the "a" variable is freed for the second time, the "b" variable is actually freed for the memory manager but not from a code point of view. By using this double freed variable, an attacker could read memory or run arbitrary code.

As for a use after free, static analysers should save the state of any pointers (allocated, freed) and throw warnings if a freed pointer is freed again.

2.3.8 Vulnerability 8 – Format string

```
1  #include <stdio.h>
2  #include <string.h>
3  
4  char password[14] = "VESSEDIAh2020";
5  
6  static int auth(char *user, char *pass) {
7      int is_admin = 0;
8  
9      if (!strcmp(pass, password)) {
10         is_admin = 1;
11     }
12     return is_admin;
13 }
14 
15 int   main(int argc, char **argv) {
16     if (argc < 3) {
17         puts("Please input username and password to authentify yourself.");
18         return 1;
19     }
20     printf("* Authentifying user ");
21     printf(argv[1]);
22     if (auth(argv[1], argv[2]) != 1) {
23         puts("\n- Incorrect user or password");
24         return 1;
25     }
26     puts("\n+ Welcome to the admin panel.");
27     /// ...
28     return 0;
29 }
```

Format string vulnerability is part of the most dangerous ones because it gives an attacker the ability to read or write anywhere in memory. As the *printf* family functions can take an arbitrary number of parameters, they're passed onto the stack. So, when a buffer is passed without the "%s" modifier, and contains any other modifiers, so these modifiers will be interpreted. For example, a "%x" modifier will print the first value on the stack in hexadecimal, leading to an arbitrary read condition or arbitrary code execution if the "%n" modifier is used.

This example illustrates a vulnerable authentication program due to the improper use of *printf* function at line 21. There are multiple ways of exploiting this program. Firstly, a user could give a format string containing the "%s" modifier to read memory and then printing the *password* variable. Secondly, one could overwrite the *is_admin* variable using the "%n" modifier that allows to write in memory. These two techniques could easily let attacker access the admin panel.

A warning should be thrown if either none modifier is passed a parameter or if the number of modifiers does not match the number of subsequent parameters.

2.4 Metrics

In order to rate the effectiveness of the tested tools, these benchmarks will be based on the following metrics:

- **Detection:** a Boolean to check if the vulnerability is detected or not;
- **Calculation time**: How long does the tool take to produce its results;
- **False positive**: How many thrown warnings are actually false positive.

Chapter 3 Test environment

3.1 Platform

The platform used to perform these benchmarks is a Linux Debian[6] 9 (Stretch) in 64-bit version.

This machine is powered by a dual-core processor and 4 GB of RAM.

3.2 Tools

The samples from these benchmarks will be analysed by three different static code analysers. Thus, Frama-C will be compare to CodeSonar and Clang analyser

3.2.1 Frama-C – Chlorine release

Frama-C is an open-source C and C++ source code analysis framework co-developed by CEA and INRIA. Frama-C is available at https://frama-c.com.

The two main plugins for Frama-C are EVA and WP. They are presented in [D4.2]. The WP plugin implements a Hoare calculus in order to prove the correctness of ACSL specifications of the source code. Thus, the WP plugin is semi-automatic and it requires to write complete specification in ACSL to guarantee that the source code is compliant to its specification. As explained in chapter 2.1, these benchmarks will focus on a more automatic approach.

Once properly configured, the EVA plugin runs automatically. It is based on abstract interpretation. The EVA plugin is exhaustive and produce the complete list of potential run-time errors. That is to say, if no alarm is emitted for an operation in the source code, then this operation is guaranteed not to cause a run-time error.

3.2.2 CodeSonar – Version 5

CodeSonar is a commercial static analysis tool developed by GrammaTech. It aims to scan source code to detect most well known vulnerabilities in C, C++ or Java languages. CodeSonar needs a compiling project to be able to work or can operate on a binary directly. By compiling the project, CodeSonar is able to build an abstract syntax tree and a control-flow graph of the analysed software. It then uses a symbolic execution engine to explore paths of the control-flow graph in order to find particular properties or patterns that indicate defects.

CodeSonar is available for download at https://grammatech.com/products/codesonar.

3.2.3 Clang – Version 3.8.1

The Clang static analyser is a free source code analysis tool that search for bugs in C, C++ and Objective-C programs. Clang works on the intermediate compiled state of the program and uses a path sensitive analysis technique. Path sensitive analysis is a technique that explores all the possible branches in code and records the code paths that might lead to bad or undefined behaviour.

Clang static analyser is available for download at https://clang-analyzer.llvm.org.

[6] http://debian.org

Chapter 4 Samples

Samples from the benchmarks are described below. For each of the eight implemented vulnerabilities, a simple and a more complex example are provided. Each sample is written in C language.

The vulnerability implemented in each source code is explained and results expected from static analysers are presented.

4.1 Sample 1 – Stack buffer overflow

4.1.1 Simple example

The stack buffer overflow vulnerability is described at § **2.3.1.**

Static analysers should be able to detect this kind of vulnerability in its simplest form. Ideally, the analyser should assert that a destination buffer is large enough to contain the source data.

4.1.2 Complex example – basic_messaging

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/tree/master/challenges/basic_messaging.

Source code presentation

```c
void cgc_list_unread_messages( pmessage_manager pmm )
{
	pmessage walker = NULL;
	cgc_size_t size = 0;
	unsigned char count = 0;

	if ( pmm == NULL ) {
		return;
	}
	walker = pmm->root;

	while ( walker ) {
		if ( walker->cgc_read == 0 ) {
			count++;
		}
		walker= walker->next;
	}

	if ( count == 0 ) {
		return;
	}

	cgc_puts("Unread messages:\n");

	// Calculate size
	// Message Text
	size = count * MESSAGE_LENGTH;
	// Message border
	size += count * (72);
	// Message id and trailing newline
	// "###:  "
	size += count * 8;

34         char data[size];
35   }
```

This sample illustrates a text messaging application. The *count* variable, used to store the number of unread messages is an unsigned char. So, if more than 255 messages are waiting to be displayed this counter will overflow. The counter is used to calculate the size of the buffer necessary to display the messages. A counter that has overflowed will result in the allocation of a buffer that is too small to contain the entire stream of messages.

The particularity of this sample is that the buffer overflow condition is nested in a loop. This makes the vulnerability discovering more challenging in the context of a static analysis.

Static analysers should denote that a buffer overflow may occur on the *count* variable if at least 255 iterations of the loop are done and the condition is satisfied.

Expected Results

A warning should be thrown in *service.c* at lines 445-450, which is where the overflow occurs.

4.2 Sample 2 – Heap buffer overflow

4.2.1 Simple example

The heap buffer overflow vulnerability is described at **§2.3.2**.

A static analysis should detect overflows by comparing number of written bytes with the initial allocation size.

4.2.2 Complex example – simplenote

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/simplenote.

Source code presentation

```
1    note_t* cgc_append_note(note_t* note, char* data)
2    {
3      if (!note || !data)
4        return ((void *)0);
5
6      if (strlen(data) + note->size + 1 > note->cap)
7      {
8    #ifdef PATCHED
9        note->cap = (strlen(data) + note->size + 1) * 2;
10   #else
11       note->cap *= 2;
12   #endif
13       note->buf = realloc(note->buf, note->cap);
14       ({ if (note->buf == ((void *)0)) ({ fprintf(2, "ERROR %s:%d:\t" "bad alloc"
15   "\n", "src/main.c", 124); cgc_exit(1); }); });
16     }
17
18     strcat(note->buf, data);
19     note->size += strlen(data);
20     return note;
21   }
```

The vulnerability exposed in this note manager is a heap buffer overflow caused by an improper reallocation. In fact, the *cap* variable is multiplied by 2 (l.11) without considering the current size. So, if the current note size is small and the length of the new data to append is greater than twice the current size of it, a heap buffer overflow condition is created. This can lead to information leakage or code execution.

A static analysis should reveal a potential heap buffer overflow vulnerability by asserting that the length of data to add is strictly inferior to the current note size. Static analysers should be able to “follow” values changes. In this particular case, syntax-based analysers won’t be able to detect the vulnerability.

Expected Results

A warning should be thrown in *main.c* at line 295, on the call to *strcat* function.

4.3 Sample 3 – Null pointer dereference

4.3.1 *Simple example*

The null pointer dereference vulnerability is described at **§2.3.3**.

Static analysers should warn if any pointer having an address inferior to 4096 is dereferenced.

4.3.2 *Complex example – BitBlaster*

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/BitBlaster.

Source code presentation

```
 1  int main(int cgc_argc, char *cgc_argv[])
 2  {
 3    uint8_t temp;
 4
 5    cgc_initBoard();
 6
 7    while (!cgc_isWinner())
 8    {
 9      if (cgc_receive_all(0, &temp, 1) != 1)
10      {
11        cgc_printBoard();
12        return (0);
13      }
14      if (temp == END_CHAR)
15      {
16        cgc_printBoard();
17        return (0);
18      }
19      cgc_makeMove(temp);
20    }
21
22  #ifdef PATCHED
23  #else
24    ((int (*)())0)();
25  #endif
26
27    cgc_transmit_all(1, "You Win\n", 8);
28  }
```

The vulnerability exposed in this board game is a null-pointer dereference caused by the explicit call to a zero function pointer at line 24. This sample shows an obvious and not so realistic case of a null-pointer dereference because of the explicit call to a function at address 0. However, it's an interesting case for static analysis to see how the detection is done.

The analysis should reveal access to an unmapped region or a null-pointer dereference and point out that this can only happen if the function *cgc_isWinner* return "true".

Expected Results

A warning should be thrown in *main.c* at line 207. This is where the null pointer dereference occurs.

4.4 Sample 4 – Use after free

4.4.1 *Simple example*

The use after free vulnerability is described at **§ 2.3.4**

Static analysers should check the state of any pointers (allocated, freed) and throw warnings if a dangling pointer is used.

4.4.2 *Complex example – Movie_Rental_Service*

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/Movie_Rental_Service.

Source code presentation

```
void cgc_remove_movie()
{
  char buf[256];
  unsigned int num_movies = 0, movie_id;
  movie_node_t *node;

  /* Movie list (full) */
  cgc_printf("\nMovies (Full)\n--------------\n");
  for (node = movies_full; node != NULL; node = node->next)
  {
    num_movies++;
    node->movie->print_info(num_movies, node->movie);
  }
  cgc_printf("--------------\n%d movie(s)\n", num_movies);
  if (num_movies == 0)
  {
    cgc_printf("[ERROR] Movie list is empty.\n");
    return;
  }

  /* Get and validate the index */
  while (1)
  {
    cgc_printf("Enter movie id: ");
    if (cgc_readuntil(STDIN, buf, sizeof(buf), '\n') < 0)
      return;
    movie_id = cgc_strtoul(buf, NULL, 10);
    if (movie_id >= 1 && movie_id <= num_movies)
      break;
    cgc_printf("[ERROR] Invalid movie id. Try again.\n");
  }

  node = cgc_movie_find_by_id(movies_full, movie_id);
  movie_t *movie = node->movie;

#if PATCHED
  if (cgc_movie_delete(&movies_full, movie_id) != 0 ||
cgc_movie_delete(&movies_rented, movie_id) != 0)
#else
  if (cgc_movie_delete(&movies_full, movie_id) != 0)
#endif
    cgc_printf("[ERROR] Failed to remove.\n");
  else
  {
    cgc_g_num_movies--;
    cgc_free_movie(movie);
    cgc_printf("Successfully removed the movie!\n");
  }
}
```

This sample is a movie rental service which exposes a use-after-free vulnerability that can be used to control the process execution flow. In this case, the *cgc_movie_delete* function removes elements from the global list but not from current rented movies list. Thereby, other functions will continue to use these elements even if they are freed, causing a buffer overflow. By overflowing the heap layout, attacker may override a function pointer to execute arbitrary code.

Use-after-free vulnerabilities are difficult to discover with static analyses especially for syntactic analysers. Using a precompiled state of the program (like *Clang* or *Valgrind*) or doing symbolic execution will be much more efficient.

Expected Results

A warning should be thrown in *main.c* at either line 320 (allocate), or line 481 (free).

4.5 Sample 5 – Off by one

4.5.1 Simple example

The off by one vulnerability is described at **§2.3.5**.

As for a classical stack buffer overflow, static analysers should throw an out of bounds write error if any byte is written outside the buffer's limits.

4.5.2 Complex example – Lazybox

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/Lazybox.

Source code presentation

```
#define MAX_CMD_HISTORY (16)
#define MAX_CMD_LEN (256)
#define MAX_ARGS (8)
#define MAX_ARGLEN (63)
typedef struct _command {
      uint32_t argc;
      char argv[8][64];
} Command, *pCommand;

typedef struct _env {
        uint8_t NumCommandHistory;
        char CommandHistory[MAX_CMD_HISTORY][MAX_CMD_LEN];
        char User[32];
        char Group[32];
} environment;
```

```
void cgc_PrependCommandHistory(char *buf) {
	uint8_t i;

	cgc_ENV.NumCommandHistory = 0;
#ifdef PATCHED_2
	for (i = MAX_CMD_HISTORY-1; i > 0; i--) {
#else
	for (i = MAX_CMD_HISTORY; i > 0; i--) {
#endif
		if (cgc_ENV.CommandHistory[i-1][0] != '\0') {
			strcpy(cgc_ENV.CommandHistory[i], cgc_ENV.CommandHistory[i-
1]);
			if (cgc_ENV.NumCommandHistory == 0) {
				cgc_ENV.NumCommandHistory = i;
			}
		}
	}
	strcpy(cgc_ENV.CommandHistory[0], buf);
	cgc_ENV.NumCommandHistory++;
}
```

In this sample, the *CommandHistory* buffer is used to store command line history and can contain at most 15 entries. However, in the "for" loop, the increment variable "i" is initialised with the *MAX_CMD_HISTORY* value. This leads to an off-by-one bug, creating a buffer overflow condition and can result in arbitrary code execution.

Static analysers should be able to detect this vulnerability by comparing the *MAX_CMD_HISTORY* value with the buffer index accessor.

Expected Results

A warning should be thrown in *shell.c* at line 284. The overflow caused by the off-by-one bug is happening here.

4.6 Sample 6 – Uninitialised variable

4.6.1 Simple example

The uninitialised variable vulnerability is described at **§2.3.6**.

A static analysis should reveal all uninitialised variables that are used, even if the potential initialization is executed only in a conditional block.

4.6.2 Complex example – HackMan

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/HackMan.

Source code presentation

```
typedef struct hackman_state {
  void (*quit_handler) (void);
  void (*new_challenge_handler) (struct hackman_state *);
  char word[20];
  char progress[20];
  unsigned int num_tries;
} hackman_state_t;
```

```
void cgc_play_game()
{
#if PATCHED
  hackman_state_t h_state = {0};
#else
  hackman_state_t h_state;
#endif
  char buf[128];
  int i, found, error;

  while (1)
  {
    error = 0;
    if (win || cgc_strlen(h_state.word) == 0)
      goto new_chal;
    cgc_fdprintf(STDOUT, "[[[ Your challenge: %s ]]]\n", h_state.progress);
    cgc_fdprintf(STDOUT, "Guess a letter: ");

    switch (cgc_parse_input(buf))
    {
      case NEW_CHALLENGE:
new_chal:
        if (h_state.new_challenge_handler == NULL)
          h_state.new_challenge_handler = cgc_new_challenge;
        if (h_state.quit_handler == NULL)
          h_state.quit_handler = cgc_quit_game;
        h_state.new_challenge_handler(&h_state);
        win = 0;
        continue;
        break;
      case RESET:
        h_state.num_tries = 0;
        memset(h_state.progress, '_', strlen(h_state.word));
        cgc_fdprintf(STDOUT, "\n^^^^^ RESET ^^^^^\n\n");
        continue;
        break;
      case QUIT:
        if (h_state.quit_handler == NULL)
          h_state.quit_handler = cgc_quit_game;
        h_state.quit_handler();
        break;
      case ERROR:
        error = 1;
        break;
      default:
        h_state.num_tries++;
        break;
    } . . .

      }

}
```

The vulnerability in this sample is caused by the use of an uninitialised variable. Indeed, in the *cgc_play_game* function, *new_challenge_handler* and *quit_handler* variables are initialised only if they are NULL. Later in this sample, user is asked to type in a comment in a large buffer, thus, by controlling a part of the stack, attacker can control the two uninitialised function pointers named later and execute arbitrary code.

Static analyser should warn the user when a non-initialised variable is used and that there is no conditional or unconditional initialization of it around its declaration. Ideally, static analysers should follow variable values and detect these vulnerabilities even if they are part of a structure and should handle conditional block. If possible, it should indicate if a large buffer is allocated on the stack.

Expected Results

A warning should be thrown in *main.c* at line 185. This is where the declaration of the uninitialised variable is.

4.7 Sample 7 – Double free

4.7.1 Simple example

The double free vulnerability is described at **§2.3.7**.

As for a use after free, static analysers should save the state of any pointers (allocated, freed) and throw warnings if a freed pointer is freed again.

4.7.2 Complex example – KTY_Pretty_Printer

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/tree/master/challenges/KTY_Pretty_Printer.

Source code presentation

```c
char* cgc_parse_item(kty_item_t *item, char *str)
{
  char c;
  if (item && str)
  {
    c = str[0];
    if (cgc_strncmp(str, "true", 4) == 0)
    {
      item->type = KTY_BOOLEAN;
      item->item.i_bool = 1;
      return str + 4;
    }
    if (cgc_strncmp(str, "false", 5) == 0)
    {
      item->type = KTY_BOOLEAN;
      item->item.i_bool = 0;
      return str + 5;
    }
    if (cgc_strncmp(str, "null", 4) == 0)
    {
      item->type = KTY_NULL;
      return str + 4;
    }
    if (cgc_strncmp(str, "=^.^=", 5) == 0)
    // {
      item->type = KTY_CAT;
      return str + 5;
    }

    switch (c)
    {
      case '[':
        return cgc_parse_array(item, str);
      case '{':
        return cgc_parse_object(item, str);
      case '-': case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9': case '+':
        return cgc_parse_number(item, str);
      case '\"':
        return cgc_parse_string(item, str);
    }
  }
#if PATCHED
  if (item)
    item->type = KTY_NULL;
#endif
  return NULL;
}
```

The above sample contains a double free condition caused by a logical error. In fact, the *item->type* variable remains uninitialised and not set to *NULL* if the string passed to the *cgc_parse_item* function is not a valid type. Later in the code, the *item->type* variable is freed even if it has already been. This can result in a memory leak or arbitrary code execution.

Static analysis should throw a warning if an already freed variable is passed to the *free* function and advise the user that setting the *NULL* value after *free*ing variables is a good practice to avoid this problem.

Expected Results

A warning should be thrown in *kty.c* at line 579. This is where the double free occurs when conditions are all satisfied.

4.8 Sample 8 – Format string

4.8.1 Simple example

The format string vulnerability is described at **§2.3.8**.

A warning should be thrown if either none modifier is passed a parameter or if the number of modifiers does not match the number of subsequent parameters.

4.8.2 Complex example – Movie_Rental_Service_Redux

The source code of this sample is available at the following address: https://github.com/trailofbits/cb-multios/blob/master/challenges/Movie_Rental_Service_Redux.

Source code presentation

```
void cgc_print_genres()
{
    int i;
    for (i = 0; i < g_num_genres; i++) {
#ifndef PATCHED
        printf("\n %d. ", i+1);
        printf(g_all_genres[i]);
#else
        printf("\n %d. %s", i+1, g_all_genres[i]);
#endif
    }
    printf("\n");
}
```

The vulnerability present in this sample is quite simple to detect, it’s an improper use of the *printf* function. The *printf* function, called with a unique parameter, will pop values from the stack because of the *va_args* mechanism. Thus, if the user is able to control both passed modifiers and a stack area, he will be able to read or write almost anything in the process memory.

This issue is probably the simplest to detect. In fact, many compilers warn about this vulnerability if the mistake is detected at compile time. For a static analyser, a warning should be thrown if a function of the *printf* family is used without a matching number of modifiers and parameters or if only one parameter is passed.

Expected Results

A warning should be thrown in *cmdb_backend.c* at line 288. The *printf* function is called without modifiers.

Chapter 5 Summary and Conclusion

The goal of these benchmarks is to challenge the effectiveness of Frama-C in a security evaluation context. As most of analysed code in security evaluation are “legacy” code with no ACSL specification, focus was placed on an automatic use of the analyser, in particular with the EVA plugin.

Tests cases are based on eight of well-known vulnerabilities in C and illustrated by DARPA CGC challenges. These vulnerabilities are covering most of the main security issues found in software written in C. Even now, they are still widespread. Innovative tools are required to find efficient ways to detect them. Test cases are both based on custom basic examples and more realistic complex ones. Indeed, basic examples are used as a reference to check if a class of vulnerability can be detected by the tools, while realistic examples are used to test the detection limits of the analysis.

Two other static analysers were selected to compare their results with Frama-C: CodeSonar (commercial) and Clang analyser (open-source). Both of them are well-known and mature tools, used by industries. Results of these benchmarks will be based on three metrics: the calculation time, the detection ratio and the number of false positives. These results will be presented in [D4.5] in December 2019 (M36).

List of Abbreviations

Abbreviation	Translation
CGC	Cyber Grand Challenge
DARPA	Defense Advanced Research Project Agency
IoT	Internet of Things
EVA	Evolved Value Analysis
CWE	Common Weakness Enumeration
ACSL	ANSI / ISO C Specification Language