



D1.7

Vulnerability discovery methodology

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

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D1.7 / 1.0
Work package contributing to the deliverable:	WP 1
Due date:	Dec 2018 - M24
Actual submission date:	22 nd January, 2019

Responsible organisation:	AMO
Editor:	Cédric BERTHION
Dissemination level:	PU
Revision:	1.0

Abstract:	This document is a methodology to detail how to use Frama-C to perform a security audit of C source code. It is focus on specific constraints faced by security evaluators: time constraint and lacks of previous knowledge of the audited source code.
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

Nicolas ZILIO (AMO)

Cédric BERTHION (AMO)

DRAFT

Disclaimer

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author’s view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability. This document has gone through the consortiums internal review process and is still subject to the review of the European Commission. Updates to the content may be made at a later stage.

Executive Summary

This document first presents a comprehensive review on C source code auditing as experienced by security auditors, facing specific constraints: time constraint and lacks of previous knowledge of the audited source code. Then, a methodology detailing how to use Frama-C to perform a security audit of C source code, given the previous constraints, is presented.

DRAFT

Contents

Chapter 1 Introduction	1
1.1 VESSEDIA motivation and background	1
1.2 Role of the deliverable	1
1.3 Structure of the document	1
1.4 Related deliverables	2
Chapter 2 Security evaluation by source code auditing	3
2.1 Code analysis in security evaluation	3
2.1.1 What is it and how it differs from code review in development?	3
2.1.2 It should be all about context	4
2.1.3 On using automation tools for security review	5
2.1.4 On the conduct of a source code security evaluation	5
2.1.5 On the methodology used	6
2.2 A general methodology for code analysis	7
2.2.1 Requirements for the reviewer	7
2.2.2 Methodology overview	7
2.2.3 Methodology phases	7
2.2.4 Quick Summary	17
Chapter 3 Most Common Vulnerabilities in C	18
3.1 Technical background	18
3.2 C language intrinsic vulnerabilities	23
3.2.1 Buffer Overflow	23
3.2.2 Null pointer dereference	28
3.2.3 Uninitialized variable utilization	30
3.2.4 Double free	31
3.2.5 Use-after-free	35
3.2.6 Integer Overflow	36
3.2.7 Off-by-one	37
3.2.8 Format String	38
3.2.9 Type confusion	40
3.3 Cryptographic vulnerabilities	42
3.3.1 Non-respect to cryptographic standards	42
3.3.2 Misuse of cryptographic algorithms	42
3.4 C vulnerabilities depending on the environment	42
3.4.1 Race condition	42
3.4.2 Path manipulation	44

3.4.3	SQL Injection.....	45
3.4.4	Command Injection.....	46
3.4.5	Logic bugs.....	48
3.4.6	Contextual vulnerabilities.....	49
Chapter 4	On using Frama-C within the proposed methodology.....	51
4.1	What is Frama-C?.....	51
4.1.1	Description.....	51
4.1.2	Frama-C's intended use.....	52
4.1.3	A brief discussion on using Frama-C for security code review.....	52
4.2	Integration of the modified Frama-C into the proposed methodology.....	53
4.2.1	Using Frama-C on the automated review part.....	53
4.2.2	Using Frama-C on the manual review part of the analysis phase.....	60
Chapter 5	Applying Frama-C's use on an example.....	62
5.1	Choosing a sample.....	62
5.2	Quick discovery phase.....	62
5.3	Review phase.....	63
5.3.1	Automated review.....	63
5.3.2	Manual review phase.....	67
Chapter 6	Summary and Conclusion.....	69
Chapter 7	List of Abbreviations.....	70
Chapter 8	Bibliography.....	71
Chapter 9	Annex.....	72

List of Figures

Figure 1: proposed methodology flow	7
Figure 2: discovery phase sub-phases.....	8
Figure 3: review sub-phases	10
Figure 4: vulnerability analysis sub-phases.....	13
Figure 5: closure phase.....	15
Figure 6 : source code methodology summary.....	17
Figure 7: memory layout of a program	19
Figure 8: stack evolution in the previous program	20
Figure 9: heap layout	21
Figure 10: free of a chunk	21
Figure 11: reallocating a chunk	22
Figure 12: physical address layout.....	22
Figure 13: a pointer in memory	23
Figure 14: stack layout after variables declaration.....	24
Figure 15: stack manipulation when writing data in the buffer	25
Figure 16: pass variable overwrite	25
Figure 17: privilege escalation without the correct password.....	26
Figure 18 : heap overflow exploitation from the previous program	27
Figure 19: control flow hijack due to BSS overflow.....	28
Figure 20: bss overflow to control flow hijack	28
Figure 21: segmentation fault due to null pointer dereference	30
Figure 22: uninitialized variable utilisation	31
Figure 23: double free vulnerability leveraged into an identity theft	34
Figure 24: use-after-free vulnerability leveraged into identity theft.....	35
Figure 25: execution of the off-by-one program.....	38
Figure 26: stack layout when calling printf.....	38
Figure 27: internal working of printf	39
Figure 28: format string vulnerability used to read secret file.....	40
Figure 29: format string vulnerability used to parse the stack.....	40
Figure 30: example of type confusion.....	41
Figure 31: race condition illustration.....	44
Figure 32: path manipulation example	45
Figure 33: elevating its privileges through command injection.....	48
Figure 34: logic bug leading to wrong privileges given to the user.....	49
Figure 35 : accessing red alarms tab within the GUI of Frama-c	57

Figure 36 : using Impact plugin in the graphical interface of Frama-C (green statements are the one highlighted by Impact plugin)	60
Figure 37 : using Zones to highlight statements (in pink) that define the value of huhu variable in the printf function	60

List of Tables

Table 1: CVE count in 2017 (source: http://cppcheck.sourceforge.net/)	4
Table 2: examples of commonly used security scanners.....	11
Table 3: list of potentially vulnerable libc functions	12
Table 4: matrix giving risk score	14
Table 5: impact score determination	14
Table 6: exploitability score determination.....	14
Table 7 : example of a vulnerability presentation that could be found in the report.....	15

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 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 reviews the process of source code auditing in security evaluation, then describes how security evaluators should use and interact with the VESSEDIA tools and plug-ins to discover common C vulnerabilities, as declared in the work of Task 1.6 inside WP1. An example of using such a methodology is finally presented.

1.3 Structure of the document

The document can be divided into 5 major parts:

Chapter 2 describes the structure of a source code audit, common methodologies used, and the constraints applied to such an audit. This chapter should be considered apart from the D4.2 report's methodology, which focuses on a generic evaluation methodology. Indeed, it is considered in the latter that the final product is available and can be tested against real attacks. Common steps like risks analysis are also different due to the fact global information available to the reviewer is not the same in those two cases. However, in the case the final product is also available; this report methodology should be considered like a sub step of the generic methodology presented in D4.2.

Common C vulnerabilities and their implication are presented within Chapter 3.

In Chapter 4 we specify the security evaluation methodology to be applied with Frama-C to discover vulnerabilities.

Such a methodology on a known vulnerable app is then applied and compared to a classical approach within Chapter 5.

A Conclusion on limits and adequacy of the methodologies is done in Chapter 6.

1.4 Related deliverables

D1.1 – Security requirements for connected medium security-critical applications

D1.5-a – Vulnerabilities taxonomy

D1.5-b – Analyses choice methodology report

D4.2 – VESSEDIA Approach for security evaluation

DRAFT

Chapter 2 Security evaluation by source code auditing

This paragraph introduces the concept of security source code auditing. The idea is to highlight the process, its limitations and also the constraints that a reviewer might encounter during an audit. It will especially serve as an introduction point for the next chapter.

2.1 Code analysis in security evaluation

2.1.1 *What is it and how it differs from code review in development?*

Source code auditing is a mean by which a security auditor examines and analyses program source code for potential vulnerabilities or flaws. Its main objectives are to find potential security problems or exploitable vulnerabilities in the program, then quantify them given possible scenarios and propose solutions to it. Often, the source code audit is a complementary approach to other audits such as penetration tests and permits often to reveal security flaws that does not appear directly while doing other types of audits. Indeed, the auditor is normally here in the possession of the real backbone of the program, that should let him dive deep into his functioning. However, source code auditing does not provide a view on possible security architecture around or another security flaws mitigations tools shipped with the end program. Furthermore, some security missions only ask for a review based on the output from the analysis tools.

Source code analysis in security evaluation is far different than source code analysis in code development/review. Indeed, the latter focuses on checking coding standards (does everyone use the same programming style? Are there multiple operations on the same line?), quality of code, reducing its complexity, improving its performance and finally ensuring the absence of functional problems. The final goal is so to produce a “better code” than the previous one.

Regarding the time those reviews should be held, a development code review should occur regularly at every step of the software development life cycle. This will indeed insure that a “better” code is produced at the end. On the contrary, security review often occurs at a later point in the development process. Indeed, the code should normally “work” before the review is done, and the whole context completely defined (more on that is given later). Its conduct is so often done when a new big step of the SDLC is to be finished; like validation, pre-production and production steps. Especially, in the case those two types of reviews are both to be held by one editor, development reviews are done at a Capability Maturity Model¹ (CMM) level of 2 (development process can be repeatable) or 3 (development process is defined), while security reviews often occur when the CMM level reaches 4 (development process is managed) or 5 (development process is managed and optimized).

The reader may ask himself about the following: “- But, by ensuring a better quality of the code, and so the reduction of the bug number while doing development review, security source code analysis appears useless. What is the reason to do them so?”. Let’s consider the following scenario: the code to be analyzed is a highly critical client/server application, and so it has been proven that the client always sends packets of a given size to the server. As such, in order to improve performance, no checks are implemented on the server side regarding the size of packets. More, those packets are stored in a temporary buffer before their processing by the server. Thus, the server just reads the already defined size of packets from this buffer to treat them. There is so no bug on the application and while packets are sent by the client, the server functioning is in nominal mode. However, communications are not authenticated. Let’s now consider an attacker that mimics the client, and starts sending packets of arbitrary sizes. When the server will treat the packets from the temporary buffer, the parsing may result in a bad

¹ “Managing the Software Process”, Watts Humphrey

interpretation of fields that may lead to a crash of the server, which is critical. There is here a security problem, which is the possibility of “Deny of service”, or even “arbitrary code execution”, by an attacker. A solution is to ensure a check on the packet size on reception of data on the server side.

Finally, the aim of a security audit is to verify that an application cannot be used in any way by a malicious user, while the development review ensures that the application can be used in the intended way.

2.1.2 It should be all about context

As denoted by the previous example, the context of the application was important to understand a potential security flaw on the application. Especially, in the case where it is not possible for an attacker to mimic a client, then the risk associated with this deny of service would be much lower.

Thus, security code review should not be simply about reviewing pure code (i.e. finding so called “bugs”), and its efficiency is tied to a global context understanding. Indeed, the idea behind security code review is to ensure that the code adequately protects its assets, and protects itself against malicious entries from its environment.

Understanding the whole context permits therefore:

- To establish the risks incurred by the application. Especially, some security failures may occur even without any functional problem in the analyzed program. One example would be an improper authentication system.
- To conduct a faster/better analysis as the priority of the audit will be to analyze paths that are potentially vulnerable to a malicious entry or that are susceptible to cause the biggest risks to the entity using the application.
- To emit better recommendations, as those could fit the context in which they are emitted for.

To sum up, security review is about finding bugs that are intrinsic to the code (and that may lead to a hijack of the normal control flow of the program), but also about finding bugs that are context dependent.

As a potential proof of this statement, here is a summary (certainly not completely accurate) of the most common types of security vulnerabilities relevant to C that were declared in 2017 (from CVE count):

Category	Amount
Buffer Errors	2530
Improper Access Control	1366
Information Leak	1426
Permissions, Privileges and Access Control problems	1196
Input validation	968

Table 1: CVE count in 2017 (source: <http://cppcheck.sourceforge.net/>)

Considering that “buffer errors”, “input validation” and “information leak” are vulnerabilities that are dependent on a pure bug from the code that means here that only 65% of vulnerabilities are resulting from the code itself. Of course, this result is subject to variations (Information leak could be the result of a functional problem instead for example) and should not be taken too seriously.

2.1.3 On using automation tools for security review

As the context of the application is really important to conduct a correct security review of one code, one has to understand that running a tool on a code is not sufficient. Indeed, tools are not able to automatically understand the context and the possible risks that may arise from the context of the application (at least, those should be modeled in the tool by a human).

However, the search for intrinsic bugs can be made much more efficient by tools than by humans. Manual code reviews are indeed slow, covering 100-200 lines per hour on average. Also, there are multiple security flaws to look for in code, a lot of data flow to keep in mind and humans can only keep a small number of them in memory between the point where vulnerability is declared and the point where this vulnerability can have an impact. Furthermore, a manual code review requires a profound understanding of the language by the reviewer. It is thus a difficult task prone to lots of errors. Source code analysis tools can search a program for hundreds of different security flaws at once, at a rate far greater than any human code review. Those tools provide so a quick method, that may overall give already good results. It is also a way to give more constant results across different analyses. Indeed, tools are not subject to tiredness or stress for example, neither to the intrinsic knowledge of a human reviewer. However, these tools don't eliminate the need for a human reviewer, as they produce both false positive and false negative results, and their results still need to be triaged.

Finally, those two manners of leading an analysis are complementary, as automatic review really helps at getting a first glance on a code for an auditor.

2.1.4 On the conduct of a source code security evaluation

There is no general guidance on how a security source code evaluation should be performed (please see next paragraph for an explanation). As such, the security auditor is heavily dependent on several factors for the run of its audit:

- What is aimed at during the security audit. For example, in certain companies, it is asked that the code is security reviewed before going into production, without however strict criteria. In such cases, an automatic security review is often asked. The idea is so to use an automatic tool to detect potential security problems, and the role of the auditor is just to interpret the results of the tool, avoid false positives, and create the associated report. In other scenarios, a deep source code audit is required, for example to make sure that an application is secure before using it into highly critical environments (e.g. military, governments, finance...). In this case, the idea is to understand deeply the context associated with the application, and to be able to emit attack scenarios and risks about already found vulnerabilities.

- What is provided for the security audit. Again, it can be really motley. Sometimes, not even a complete source code is given to the security auditor, and this source code could have been obfuscated before the audit (due to the fear of intellectual property theft). In the best scenarios, the auditor is given the whole functional and specifications documentation, a complete code, a functioning application as well as some assistance from a developer.

- What time is given to the auditor. As a reference, one can take the number of lines of code (LoC) to be treated during the time given for the audit. For example, in the evaluation of KeePass done by the European commission, there were a total of 145000 LoC, for 23 days of evaluation done by 6 auditors at the same time². This means that the evaluation was done at a rate of approximately 150 LoC per hour. In the meantime, some missions concern around 20000LoC for 5 days of work of an auditor, so around 600 lines of code to be reviewed by the auditor per hour³.

² "KeePass Password Safe, code review results report",
https://joinup.ec.europa.eu/sites/default/files/inline-files/DLV%20WP6%20-01-%20KeePass%20Code%20Review%20Results%20Report_published.pdf

³ Internal mission realized by Amossys

In practice, what drives the evaluation is the price the client or the editor wants to pay. Given a price of around 600-1000\$ a day for an evaluator in Paris, source code review is in general expensive to realize. As such, it is often to see heavy time constraints for huge code base, or poorly documented one, posed to a security auditor in general. The use of a security source code analysis tool is so often required, and this tool should provide a quick way to determine vulnerabilities.

2.1.5 On the methodology used

Often security companies argue they are using well-known baselines for security code such as:

- ITIL Version 3 Service Lifecycle for Application Support⁴
- ISO/IEC 27034⁵
- NIST SP 800-37/64⁶

All those baselines are actually aimed towards the editor. The first one basically presents how an editor should prepare a security code review, and as such give some insights of the job to be done by the reviewer. Indeed, it is for example described that an introductory meeting should be held, where the reviewer is able to understand the application context, and for that, some resources should be present. As such, we know that the reviewer should conduct an introductory meeting, if possible. Also, examples of what vulnerabilities to search for are presented for web based applications but not for C directly. All the other baselines are in fact a security management plan for the whole lifecycle of a product. In ISO, there is however a security review process. This process consists in checking that every measure taken to reduce the risks that may be exploited against the application is correctly implemented. In none of those baselines there is an in-depth process to realize a security code review, from the view point of the reviewer.

Given those baselines are not aimed towards the reviewer, the whole code review methodology is in fact a generic consensus made from practical experiences or reviewers. Based on the OWASP Code review project⁷, which defines a generic flow of the work to be done for a complete audit, two main methods of manual source code auditing have been defined. The first one is a top-down approach, which consists to start from an entry point of the program and either follow all code branches from that entry point and stop when a branch with no interest is detected. The second approach is a bottom-up approach: the auditor first establishes a list of interesting functions to audit in term of security (i.e. functions using calls to known dangerous API) or points in the code known to be influenced by an attacker, determines if a security problem could arise and then identify from those functions the code serving as entry point to determine if an attacker could indeed manipulate the vulnerable function. There are pros and cons for both methods. The first one is time consuming but covers most of the source code and provides a great understanding of how the application works. The later one is time saving and focuses on areas which are the most vulnerable, but does not follow all code branches and skips some kind of vulnerabilities like logical issues. Also, the latter can only be realized correctly if the documentation provided by the editor is detailed enough to be able to determine the dangerous areas of the code.

The choice of the right method is in practice completely influenced by the time given to the auditor and the size of the code to audit. The idea is indeed to provide the best possible analysis in a limited amount of time.

⁴ <https://www.fichier-pdf.fr/2011/06/16/itil-v3-application-support/>

⁵ <https://www.iso.org/en/standard/44378.html>

⁶ <https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final>

⁷ https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents

Given the results of the previous paragraph, one has to understand that there is no generic method for a security auditor to do a code review. Especially, in the case of high constraints given to the auditor, the security audit will often be limited to a “down” approach that is simply check for known dangerous functions used and/or use an automatic tool to find potential vulnerabilities and treat the results. In this case, the auditor won't try to understand the context around the application and won't check the security implications of the entry points given by the application.

However, in case of correct constraints given to the auditor, it is better to mix the two methods: that is, to start with the top-down approach, but while checking the entry point for attacker impact, browse with a limited depth the branches from the entry point. The idea is so to gain fast knowledge of the application internal while assessing the dangerous areas of the code, in order to gain some context.

2.2 A general methodology for code analysis

This paragraph will present a generic methodology to make a security source code analysis from a reviewer view point. First, an overview will be presented, and then each point of the methodology will be explained more in-depth. With this presentation, Frama-C integration into it will be explained in further chapters.

2.2.1 Requirements for the reviewer

The reviewer should have the following qualities in order to perform a good security code review:

- The first and the most important one is the fact that the reviewer should be proficient in the language of the application audited, in order to make sure vulnerabilities won't be omitted due to incomprehension. In the case the application uses libraries or frameworks, the reviewer should also ideally know the internals of those.
- The second one is the ability to model a system given the documentation and/or the code, and being able to represent by itself the interactions of this system.
- Finally, ideally the reviewer has a good communication skill, which will be important in case where developers need to be contacted.

2.2.2 Methodology overview

The proposed methodology is composed of 4 big steps that should take place consecutively. Below is a diagram presenting it:



Figure 1: proposed methodology flow

In the first phase, the reviewer obtains the knowledge needed to conduct the rest of the analysis. Basically, the aim is to be able to define the context, and understand what the application should normally do.

The second phase is the code review phase. It is where vulnerabilities are found within the code. The third phase, being optional depending on what required, consists in being able to quantify the found vulnerabilities, and to propose countermeasures for the vulnerabilities found. Finally, the results from the whole analysis are presented in the closure phase.

2.2.3 Methodology phases

2.2.3.1 Discovery phase

The discovery phase aims at providing a full insight of the application for the reviewer. The idea is indeed to make sure that the whole context of the application is well understood, in order to provide a better analysis at the end.

This phase can be divided into 3 to 4 sub phases which can be depicted by the following diagram:



Figure 2: discovery phase sub-phases

The first one is the phase where the reviewer becomes acquainted with the task at hand. He basically realizes a check-up of the information available, as well as the objectives of the task. The next phase is a functional review of the application, in order to understand what the application should do and what its purpose is. The aim of this phase is to determine the assets that should be protected by the application. The third one is a contextual review, in order to understand the environmental context of the application. From that, the reviewer should be able to deduce the risks that the application is exposed to. Finally, a report phase should be done. The idea is to make sure the same view is shared between the editor and the reviewer, as well as having some time to tidy up his own reasoning.

2.2.3.1.1 Functional review

The functional review of the application should let the reviewer know what are the “big functions” of the code, i.e. what are the intended objectives of the code, and its I/O. Basically, it will let him know the potential entry points that could be leveraged by an attacker.

To do so, the review team should be provided with the following documents:

- Application design document: this document presents what are the requirements before the code was written, and how those are answered. Generally, a design of the components of the code, as well as their interactions, is defined in this document.
- Functional specifications: this document presents all the wanted functionalities of the program.
- Optionally, the documentation about test cases. Those will indeed provide examples of the running code to the reviewer.

Every document should finally be read and understood.

2.2.3.1.2 Contextual review

The contextual review aims at providing the reviewer with the environment of the code and its application. The idea is to let the reviewer know the most critical parts of the code, and to know the parts of the code to investigate first. In a second time, this contextual review will permit to indicate a level of impact for the vulnerabilities found.

To realize this review, the reviewer should be provided with the following documents:

- Architecture documents: documents describing an in-depth system overview, its different sub-components, their implementation and how they interact. It is basically a design file much more technical.
- Integration documents: documents describing how the final application is integrated into a workflow, what are the other components interacting with it or that can manipulate the

flows of the reviewed application. For example, a web application can be protected upstream by a web application firewall that raises attacker's level.

- Business requirements: this document presents rapidly what are the business objectives of the application.
- Risk analysis: in well-defined SDLC, a risk analysis might be already done. This analysis so provides a full insight of the threat scenarios feared from the editor's point of view, and the potential impact in case of a realization.

Every document should be fully read and understood.

2.2.3.1.3 On mixing those two reviews

There is no obligation the two previous reviews shall be done apart from each other. The methodology described here made indeed the difference as the objectives of those two reviews are not the same. If there are documents to be received, in general, all attached documentation of a code project is sent to the reviewer. More, some reviewers prefer to take the two approaches together. It is simply a question of feeling.

2.2.3.1.4 The first report phase

During the first report phase, the reviewer should be able to write down the following:

- A description of the sensitive assets that the code should protect (it can be itself) and the impact associated with their compromise. An example of such assets is presented within the report D1.1 of VESSEDIA project.
- A description of the threats against the application. For this step, a threat modeling methodology is presented within the D1.1 or D4.2 report of VESSEDIA project.
- A description of the functional security components of the application

Given the previous points, the reviewer might be able to produce a "security map" of the application that is every threat scenario and the associated countermeasures. As such, he can normally determine how to investigate the code in the next phase.

Afterwards, it could be shared and discussed with the editor. One of important aspect here is to agree on the prioritization of the things that will get reviewed, as well as explaining the reviewer expert point of view concerning security.

2.2.3.1.5 A practical approach

Most of the time, there is no such documentation, as described in the paragraph 2.2.3.1.2, attached with a code project. Or, in most cases, the documentation is outdated regarding the current state of the project. Therefore, one of the most effective ways to get started, and arguably the most accurate, is to talk with the developers and the lead architect of the application. This should not take too much time, but just enough for the development team to share some basic information about the key security considerations and controls. At least, the reviewer should be able to determine the following points:

- Aims and functionalities of the project
- Assets that should be protected by the application
- Possible important business impacts
- Possible important technical impacts
- Definition of the attack surface
- Required security controls (implemented, regular or policies ones)

Given the answers, the reviewer should so at least be able to determine the importance of the application for the enterprise and the associated biggest risks; establish the boundaries of the application and establish potential threats and controls.

In order to do, the reviewer can use simple questions like the following:

“What type/how sensitive is the data/asset contained in the application?”:

This is a keystone to security and assessing possible risk to the application. How desirable is the information? What effect would it have on the enterprise if the information were compromised in any way?

“Is the application internal or external facing?”, “Who uses the application, and are they trusted users?”

“Where does the application host sit? Is there for example a DMZ?”

“If there are internal and external users, what are the differences from a security standpoint? How do we identify one from another? How does authorization work?”

“Are there anymore security features in your architecture?”

“How important is this application to the enterprise?”

Finally, a walkthrough of the actual running application is very helpful to give the reviewer a good idea about how the application is intended to work. Also, a brief overview of the structure of the code base and any libraries used can help the review to get started.

2.2.3.2 Review phase

The review phase consists in actually analyzing the source code for flaws. This is the technical part of the methodology. The review phase should consist of a manual investigation and also an automated one to detect flaws in the code. The following diagram presents the steps of the review phase:



Figure 3: review sub-phases

2.2.3.2.1 Automated review phase

At this step of the analysis, the reviewer should know the global context of the application, and may be able to determine the structure of the source code. He still hasn't analyzed the code. In order to get a first insight into the code and its actual security, as well as eliminating the possible intrinsic vulnerabilities that may be found in the code, a security source code analyzer can be run. Indeed, as explained in the paragraph 2.1.3, as tools will be more efficient for such a task, as finding intrinsic vulnerabilities do not require a context comprehension and are faster and more precise. This phase is so to get the tool up and running on our code base. It can sometimes need some tweaks to do so.

The source code analyzer has to be run on the application without any knowledge of it. It is what is called in the security field a “scanner”. The analyzer indeed scans the code in order to find

vulnerabilities. During the automated analysis, the reviewer can start his own manual analysis that will be described in the following paragraph.

In the case of a C/C++ code, from a practical point of view, here is a list of commonly used tools in evaluation centers:

Tool	Licensing type
Code Sonar	Commercial
IBM Appscan	Commercial
Checkmarx	Commercial
CppCheck	Free
FlawFinder	Free

Table 2: examples of commonly used security scanners

The practical point of view of this step cannot be described here, as it is heavily dependent on the tool used. For example, with Code Sonar, the methodology is the following:

- 1) Make sure that the code is almost complete and that a compilation chain can be realized
- 2) Use that compilation chain with Code Sonar (for a Makefile, the command line will look like : "codesonar make")
- 3) Wait for the analysis to be finished

2.2.3.2.2 Results triaging

Once the automated analysis is done, the reviewer should gather results and process them: the idea is to eliminate false positives and ensure true positives.

From a practical point of view, this step is often considered as a "quick-win" step. Basically, if a reviewer can determine easily if a result from the automated tool is a true positive, then he should gather it and report the vulnerability afterwards. Otherwise, especially when determining if the vulnerability is a true or false positive can't be decided easily, the reviewer should first gather the result as a work-in-progress. He could come back to it once its manual phase is done. Indeed, he will have a more in-depth understanding of the code, allowing him to make an easier decision.

2.2.3.2.3 Manual Review phase

This phase and the next big step of the methodology (e.g. the vulnerability analysis phase) can be done simultaneously. Here, as already stated in the paragraph 2.1.5, there is no predefined method to use. We will however consider that the reviewer uses a mixed approach. The idea is so to determine if a vulnerability may appear in the code given the paths taken from the possible risks that have been identified in the first phase of the methodology.

The steps are the following:

- take one of the most impactful risks identified
- identify every possible areas of the code where this risk may appear
- check if those areas are vulnerable or not
- finally, check possible entry points leading to those areas and check rapidly for vulnerabilities in the code branches taken.

Also, in the case the reviewer did not have any tool to perform the automated review of the previous paragraph, then he should perform a search on known possible vulnerable functions, and determine if their usage is secure or not. In the case the reviewer is using Linux; `grep` might be the tool to use there. Here is a list of known potential functions from the standard Linux `libc` that may have security implications (the reviewer should include here platforms specifics, like `ascii` and wide variants of multiple `libc` functions in Windows):

potentially vulnerable libc functions				
strcpy	wcscpy	strncpy	memcpy	bcopy
strcat	strncat	strncpy	strcadd	gets
sprintf	vsprintf	swprintf	vswprintf	fprintf
syslog	snprintf	vsprintf	scanf	vscanf
wscanf	fscanf	sscanf	vsscanf	vfscanf
strlen	wcslon	streadd	strecpy	strtrns
realpath	getopt	getopt_long	getwd	getchar
fgetc	getc	read	access	chown
chgrp	chmod	vfork	readlink	tmpfile
tmpnam	tempnam	mktemp	mkstemp	fopen
open	exec	execl	execlp	execl
execv	execvp	system	popen	atoi
atol	drand48	erand48	jrand48	lcong48
lrand48	mrnd48	nrnd48	random	seed48
setstate	srand	strfry	srandom	crypt
chroot	getenv	getlogin	cuserid	getpw
getpass	signal	ssignal	memalign	recv
recvfrom	recvmsg	fread	readv	strcasecmp

Table 3: list of potentially vulnerable libc functions

Finally, once the manual review of the different risks has been done, the reviewer should take again the results from the automated review, and fully qualify the vulnerabilities that were not already qualified (e.g. when it's was difficult to tell if a result was a false positive or not).

2.2.3.2.4 Reporting phase

In this phase, the idea is creating an inventory for all vulnerabilities found. To do so, for each of the vulnerability identified, the following information should be written down:

- The vulnerability type, i.e. a generic definition of it. Those generic definitions can be found in the next chapter as well as a bit of explanations for a reader to understand this vulnerability.
- The place in the code where the vulnerability can be found so the file name and the line number
- A code snippet in which the vulnerability can be found. In the case some data flow is required to fully understand the vulnerability in the code snippet for a reader, it is interesting to provide multiple code snippets explaining the data flow between the

vulnerability declaration and vulnerability impact (i.e. from where the vulnerability is in fact declared to the point where it has a real impact on the application).

2.2.3.3 Vulnerability analysis phase

This phase is a consolidation phase, so not necessary, but really interesting in a proper audit. The idea is to be able to fully determine the risk of a vulnerability found, in order for the editor to have a plan of remediation. Indeed, this plan permits to know what to prioritize to correct the application. A diagram presenting this phase is the following:



Figure 4: vulnerability analysis sub-phases

First, the exploitability of each vulnerability identified is determined. Then, a risk analysis can be realized. Finally, security countermeasures can be defined.

2.2.3.3.1 Exploitability determination phase

This phase aims at determining if a vulnerability is exploitable, that means, can an attacker leverage the vulnerabilities identified for its own profit (data theft, unauthorized access, destruction...)? Indeed, even if a vulnerability has been identified, it is not necessarily exploitable. This step should also be taken apart from the fact that an exploitable vulnerability will or will not be exploited by an attacker.

Moreover, from a business point of view, it is always interesting if the reviewer can provide some code that shows the exploitability of a vulnerability. It will provide developers with a proof of concept that has important psychological impact, that avoids their denial and becomes a proof of the job being done.

To do so, a full data flow analysis should be realized. The idea is indeed to check:

- 1) if there is an entry point that can lead to the manipulation of the variables involved in the vulnerability.
- 2) If every condition that could lead to exploitation can be met. Indeed, in some case, the vulnerability found can be only exploited if, for example, enough information can be leaked.

Once again, there is no generic practical method to determine if a vulnerability can be exploited or not. This step relies a lot on the reviewer's expertise, as it needs the capability to understand how the code behaves.

2.2.3.3.2 Risks analysis phase

Once the exploitability of a vulnerability has been identified, a risk analysis can be performed to provide a quick insight of a level of risk for this vulnerability (that will be used for the action plan afterwards the review).

The level of risk is defined as a combination of a level of impact, indicating the gravity of a produced effect in terms of security, and a level of exploitability, indicating the ease of the exploitation by an attacker. Of course, the impact level is determined given the contextual information gathered in the discovery phase, whilst the exploitability level is determined by the vulnerability analysis.

The score of risk is given by the following matrix:

Impact	low	average	high
Exploitability			
high	average	high	critical
Average			
low	low		

Table 4: matrix giving risk score

Finally, the following criteria have been selected to determine the impact score:

Level	Interpretation
Low	The exploitation does not permit the attacker to gain interesting information nor any additional privileges to compromise an asset of the system.
Average	The exploitation leads to the compromise of a non-critical asset for the system.
High	The exploitation leads to the compromise of a critical asset for the system.

Table 5: impact score determination

And here is the criteria to determine the exploitability score:

Level	Interpretation
Low	The exploitation is not feasible in the current state of the program (but it could become so in future evolutions of it)
Average	The exploitation must be realized by an expert attacker.
High	The exploitation can be done by a non-expert attacker, or can even be realized automatically by tools.

Table 6: exploitability score determination

The attentive reader may have noticed that a different approach has been taken to evaluate the risk level of a vulnerability here than how it is presented in the report D4.2. Indeed, the vulnerability cannot be tested in its final environment, and as such the final “likelihood” cannot be evaluated correctly in general. However, in the case the final application can be reviewed along with the code, then the D4.2 methodology should be followed.

2.2.3.3.3 Countermeasures definition phase

Finally, the last phase of the audit is to define potential countermeasures to a given vulnerability. Given the context of the application, the idea is to propose a measure that corrects the vulnerability with the least impact on the whole architecture and least budget. Indeed, the less difficult a measure is to be implemented, the more likely it will be implemented by the editor.

Unfortunately, there is once again no more practical method to use. In the case the vulnerability can be corrected in the code, without changing the components design, a patch can be created by the reviewer. In some cases, the solution might be to attach a COTS (component-of-the-shelf, e.g. a commercial and already existing component) component with the application.

2.2.3.3.4 Report phase

Once again, the risk analysis should be included in the report for each vulnerability found, as well as the countermeasures definition. A vulnerability can roughly be presented in the report as given below:

Risk level (with colors): Huge	Vulnerability's name	
Summary	A quick summary	
Risk	Impact level	Huge and explanations
	Exploitability level	Huge and explanations
Vulnerability location	Source file and line number	
Vulnerability description	Full description and code snippet	
Countermeasure	Countermeasure definition	

Table 7: example of a vulnerability presentation that could be found in the report

Below is also a note on how the whole report can be presented. A good layout is to provide the report for two different readers: the technical staff, which is in charge of the code, and the management staff. The management staff likes to have a quick look on statistics regarding the analysis, while the technical staff really likes in-depth analyses.

To do so, the general layout of the report can be the following:

- 1) *Introduction*
- 2) *Analysis methodology presentation*
- 3) *Quick results and statistics*
- 4) *Technical part*
 - a. *Discovery phase analysis*
 - b. *Vulnerabilities*
 - c. *Resume of vulnerabilities (basically tables with a prioritization by impact to have an almost already done risk reduction plan)*
- 5) *Conclusion*

2.2.3.4 Closure phase

The closure phase is basically the end of the review. The report is finalized, and a closure meeting is held if possible. Here is a diagram representing it :



Figure 5: closure phase

2.2.3.4.1 Final report phase

Once at this step, the report is finalized and delivered to the editor. From the layout proposed at paragraph 2.2.3.2.4, this step consists in calculating statistics for the management staff, as well as doing the resume tables for the technical staff. A quality control is realized and finally the report is ready to be delivered.

2.2.3.4.2 Ending meeting phase (optional)

Ideally, all reviews should end with an ending meeting. This meeting should present first the risks identified, as well as statistics on vulnerabilities found and costs of countermeasures. This part is known as a “management meeting part”. The idea is so to provide quick insights for the decision-makers. The second part of the meeting should be technical focused, for developers to be able to understand all vulnerabilities found. This part is so a review of every vulnerability found, possibly with a code snippet, and a quick explanation, as well as a slide on countermeasure and exploitability. Normally, it is ensured all vulnerability is understood by the technical staff of the editor.

Finally, a meeting is certainly a better approach than just giving a report to the editor. Indeed, it allows normally for a constructive review of the whole audit, where the results can be discussed and adjusted live. However, in practice, most of the time, decision-makers will hate the process unless there is no vulnerability found, and those meetings tend to be bashing meeting.

2.2.4 Quick Summary

Here is a diagram representing a quick summary of the whole methodology:

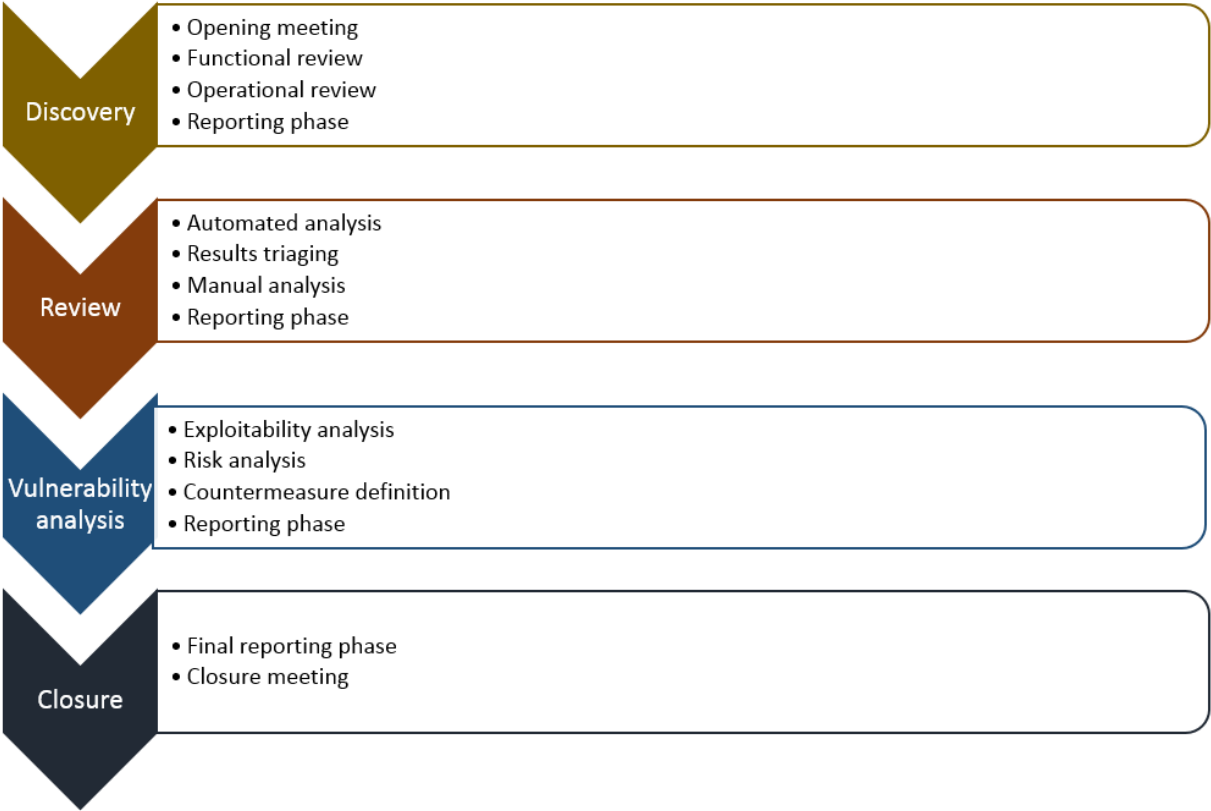


Figure 6 :source code methodology summary

Chapter 3 Most Common Vulnerabilities in C

C code is more vulnerable than other “new high-levels” language like java or C#. Indeed, it is a compiled language with a weak semantics, transformed to machine code that is run as is. This weak semantics permits especially to write code leading to undefined behaviour or that is intrinsically unsafe. Moreover, there are especially no additional security mechanisms added to the language itself, because there is no interpreter capable of determining a deviant behaviour.

In order to be able to understand the following chapters, this paragraph introduces the most common vulnerabilities that can be found in C source code, given practical examples. As such, it will be possible later to name vulnerabilities when arguing about Frama-C capabilities, and to fully understand the example analysis.

For a comprehensive list of vulnerabilities that can be found, please refer to the deliverable D1.5-a of VESSEDIA project.

3.1 Technical background

C is a compiled language, which means it is run directly by the underlying operating system as machine code. There is no interpreter for a program in C, which is able to detect potential vulnerabilities while the program is running. It also directly interacts with the underlying resources. Especially, on any computer, a program is composed of instructions, which tell what to do, and memory, where the necessary states of our program are stored for it to function normally. To fully understand, one has to imagine the case of a program which wants to switch the value of two variables, on a computer capable of handling one value at a time, it basically needs to do something like that (its instruction):

- 1) Let A a variable
- 2) Let B a variable
- 3) Save B value somewhere
- 4) Move A value into B
- 5) move the saved B value to A

There is a need to save the B variable in memory there. Moreover, in order to have this program functioning, we need to be able to read its instructions. Those are also saved into memory that will be accessed by the computer. Nowadays, the same principles apply, but at a bigger scale.

To fully understand C vulnerabilities, one has to understand that a C program needs memory, and that C programs directly interact with it. Nowadays, any memory of a program is divided basically in four parts, represented on the following diagram:

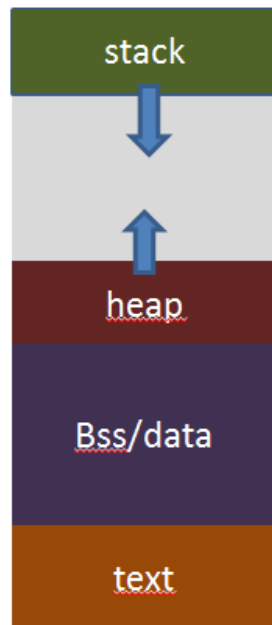


Figure 7: memory layout of a program

- There, the text section is in fact the instructions of the code.
- The static/global section is a part of memory that is related to static variables, or global ones. The memory always has the same size, whatever are the instructions of our program executed. It is also named the BSS/data section.
- The stack and the heap are the part of memory of a program that dynamically moves with the instructions executed. Basically, it was decided to have two sections here to improve the global performance of the programs.

Indeed, the stack is a part of the memory allocated by the operating system for the execution of a task. It functions like a stack of objects, that is, the last item in the stack is the first one that can be removed. The stack is used in particular when the variables are transmitted from one function to another. Thanks to the stack, the task does not have to remember the location of an item within the stack, which makes the stack much faster in its use.

On the opposite, a program needs also some memory that can be managed directly by itself, where a chunk of it can be allocated, used and freed at any moment. It is also important in the case a program need a huge memory footprint to avoid using the stack, and that might otherwise not be fast enough. This part is the heap. Of course, heap management is more complex and slower because it is necessary to know permanently which block is allocated.

In order to make sure a program won't eat all memory that is logically available on a system, the stack grows towards the heap, and vice versa. When heap and stack are superimposing themselves, then the underlying OS knows that every memory has been consumed, and it kills the process to make sure physical memory won't be filled by the program.

Nowadays, the stack is used basically to store every local variable of functions and their arguments, to provide a quick way to operate on them. To make sure a program works, some other information are also used by the stack, and those are declared by the CPU architecture. To fully explain it, let's take the following C program and the classical intel x86 architecture:

```

void foo(int j, int o){           (3)
    int i=2;                      (4)
    i = i + j + o;                (5)
    [...]

```

```

    return;                (6)
}
void main() {              (1)
    int j = 3;             (2)
    int o = 4;
    Foo(j,o);             (3)
}

```

In this example, the main function is first loaded (1), allocates two local variables `j` and `o`, then calls `foo` (3) with the `j` and `o` variables as arguments. At this moment, stack is used for memory management:

First, the argument `o` for `foo` is pushed on the stack and then the first argument `j`, then a function scope stack is created by first pushing on the stack the value of the returning address of our function. So, when the stack will be destroyed when `foo` ends, the address for the program to continue will be known. Then, the base of the stack of the main function is saved on the stack (this base is used to navigate more easily between stacks scope, and to provide a quick way to navigate between function arguments, accessed like `ebp+X`). All of this is done in (3). Finally, as `foo` function use a local variable, `i` is pushed on the stack (4). Starting at (5), the stack is fully set for the execution of function `foo`. When the function ends (6), its stack frame is basically destroyed: variable `i` is first removed from it, the base address of main function is restored, and the return value on the stack is used to go to the end of the main function and continue the program execution.

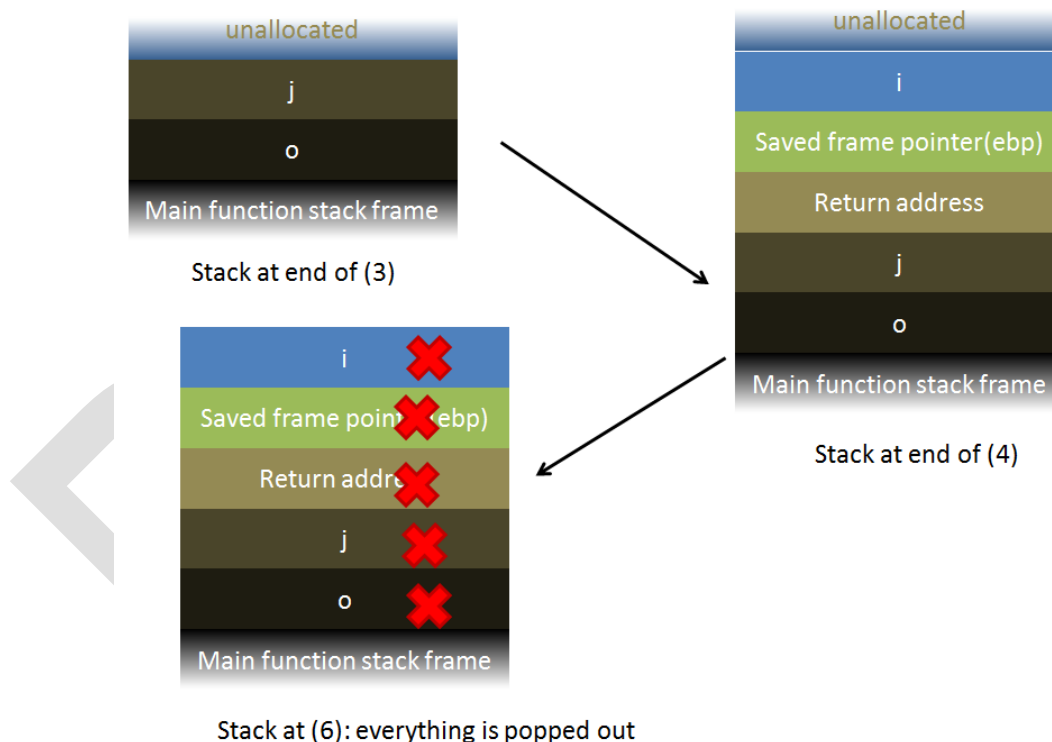


Figure 8: stack evolution in the previous program

Heap is managed with totally different mechanisms. As memory can be freed and allocated at any moment during the execution of the program, allocations are managed as chunks. A chunk is composed of two parts: a header indicating several information on the allocated data like its state and also two pointers - one to the previous allocated chunk, and one to the next allocated chunk - , and finally the allocated data. This actually creates a double linked list of the chunks. In the meantime, the same double-linked list is used for freed chunks:

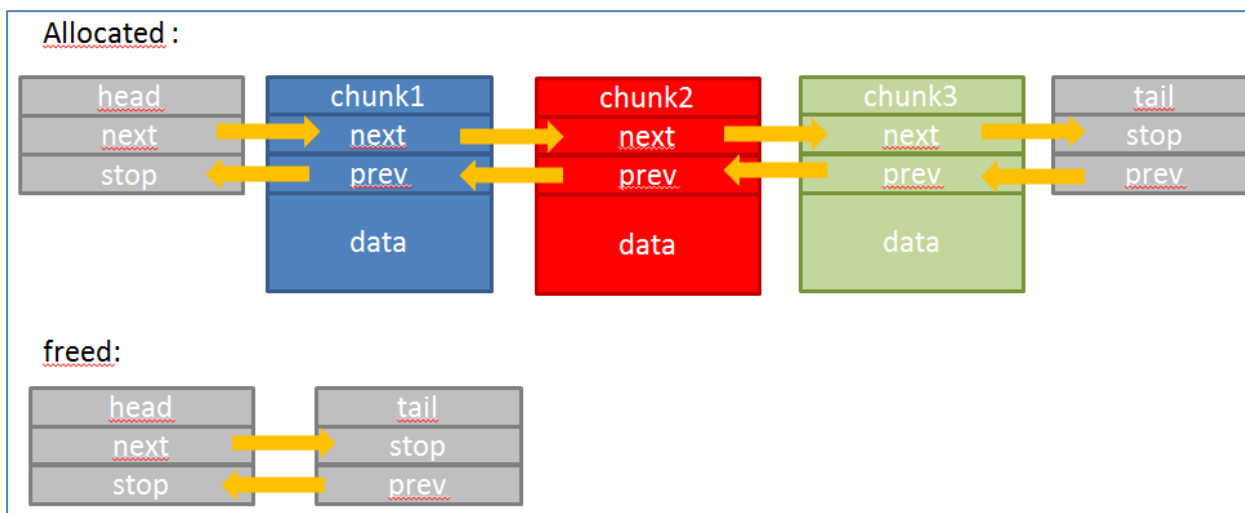


Figure 9: heap layout

When a chunk is freed, its state is passed to free and the chunk is moved from the allocated double-linked list to the freed one:

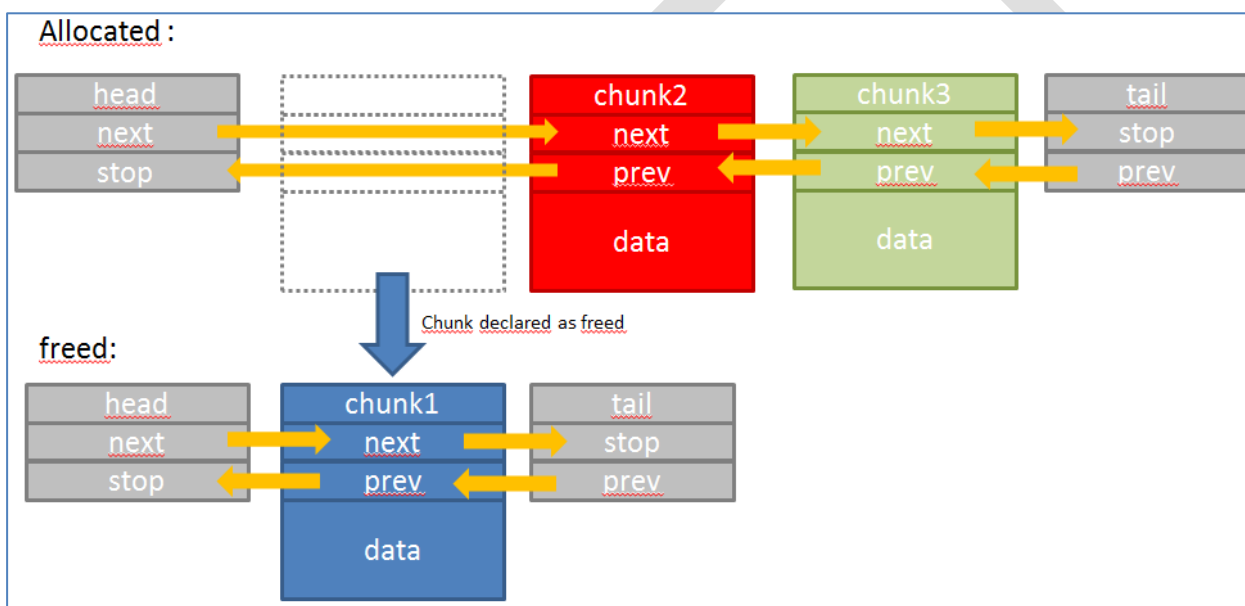


Figure 10: free of a chunk

Of course, all of this is done to have an overall better performance, as it avoids to modify the whole chunk to reassemble it with the rest of unallocated memory. When a chunk is reallocated, it is first searched if there is a freed chunk with sufficient memory space. If this is the case, then this chunk is removed from the double-linked list of freed chunks and added to the one of the allocated ones.

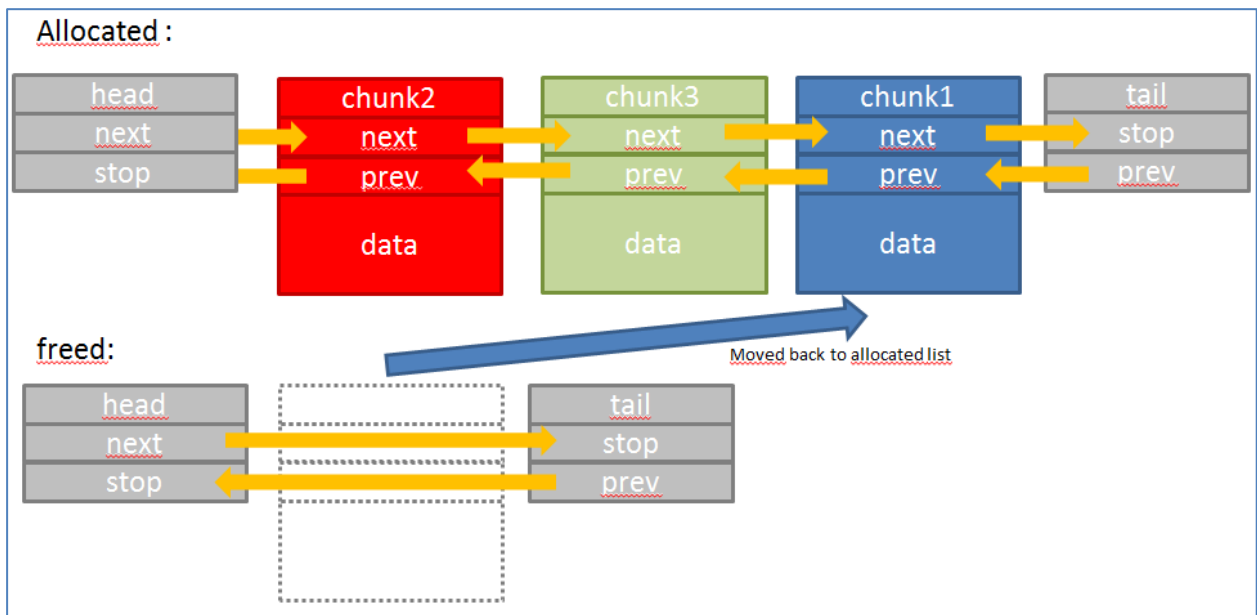


Figure 11: reallocating a chunk

If it is not the case, then multiple freed chunks can be merged together. In some memory managers, there are also additional mechanisms to treat differently the memory given the last time of use, in order to reorder the double linked lists by priority to increase performance, and also multiple freed lists depending on the original properties of the chunks. Those mechanisms are not of interest to understand the following and won't be treated here.

Finally, here is a last technical caveat of C language to fully understand vulnerabilities presented afterwards. C uses what is called pointers as a way to manage variables that is accessing them by address instead of accessing them by value. Indeed, basically the physical memory has the following layout:

address	value
0	940,2
1	145
2	3,8000
3	390144
⋮	⋮
Last address	55,212

Figure 12: physical address layout

Indeed, every cell of the physical memory contains a value. But this cell could be also accessed knowing the offset of another cell in the physical layout. For example, if there is a variable that

needs to be accessed multiple times in a program, instead of keeping a track of the value along, one solution seems to use its address in memory to finally get its value. The most attentive reader can also note that by using the stack mechanism for arguments, those are by default passed by value between functions frames. As such, a value is not propagated between different functions in C, and the solution is to use the address instead. Address behaves like real addresses, where by knowing the address of someone, you can reach that one. Pointers are variables containing the address of another variable like in the following layout:

address	value
0	940,2
1	145
2	17432
⋮	⋮
17432	390144
Last address	55,212

Figure 13: a pointer in memory

In the figure, the cell n°2 contains the address of the cell 17432, itself containing the value 390144. If *i* variable represents the latter, then the pointer is accessed by the notation `&i` (the value 17432), a pointer is dereferenced by using the notation `*`, meaning the value is accessed given the address.

3.2 C language intrinsic vulnerabilities

3.2.1 Buffer Overflow

Buffer overflows occur when it is possible in the current flow of a program for a given sized buffer to have data written above its limits. Buffer overflows can be categorized in three categories: stack-based, heap-based and BSS-based (given the piece of memory where those occur). Heap-based overflows occur when dynamically allocated memory is overflowed by filling that memory area with too much data, usually due to some sort of miscalculation by the programmer. Stack-based overflows occur when a static-sized local buffer is overflowed by attempting to store more data within the buffer than its fixed size allows. BSS-based overflows occur when global variables memory can be overwritten, otherwise due to one of the two precedent overflows, or directly by an overflow of a static sized buffer located within the BSS.

We will illustrate a classical buffer overflow occurring on the stack. Let's consider the following code:

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void)

```

```

5 {
6     int pass = 0;
7     char buff[12];
8
9     printf("\n Enter the password : \n");
10    fgets(buff,20,stdin);
11
12    if(strcmp(buff, "VessediaPWD"))
13    {
14        printf ("\n Wrong Password \n");
15    }
16    else
17    {
18        printf ("\n Correct Password \n");
19        pass = 1;
20    }
21
22    if(pass)
23    {
24        printf ("\n Root privileges given to the user \n");
25    }
26
27    return 0;
28}

```

This code does basically the following: it first allocates some memory for a pass variable, initiated to zero, holding the result of the authentication of the user, as well as a memory buffer that will hold the pass given by the user. The password is then asked for, and compared to a hardcoded value. In the case the given password is the same that the hardcoded one, then the pass variable is set to 1. In the other case, the variable is let at value 0. Finally, if the pass variable is not zero (so if authentication succeeded), then the user is accessing the privileged area.

In this example, we can see that the size of buff variable is set to 12 on line 15. On line 18, the call to fgets is done with a second parameter equal to 20. That means that this function will copy up to 20 bytes into the buff buffer, even if its size is equal to 20. So, some memory should be overwritten. To see what data is overwritten, let's represent the stack of our function. At the start, once our main function is loaded, our two variables are put on stack in this manner (we avoided here possible arguments that may have been pushed on stack):

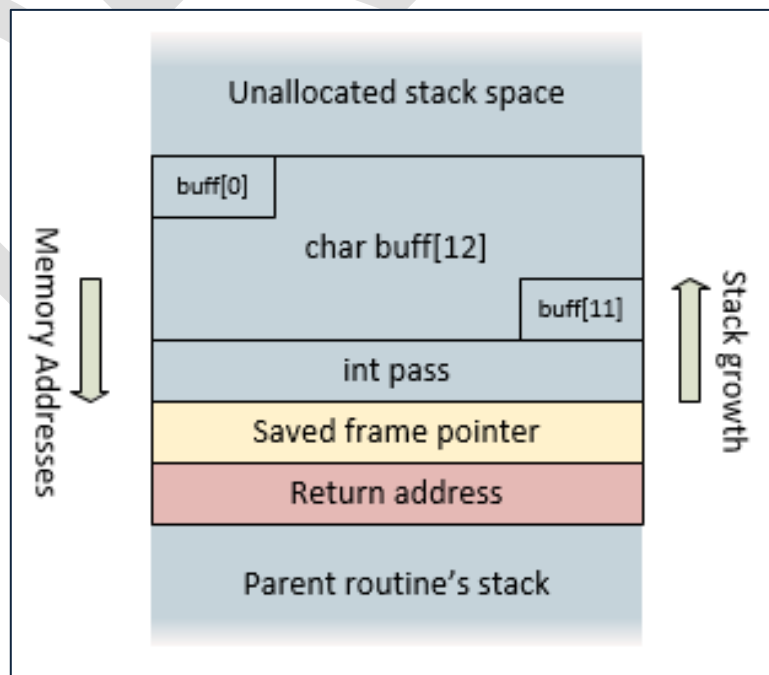


Figure 14: stack layout after variables declaration

Then, when invoking our `fgets` function, data is copied from the start of our `buff` buffer, like this:

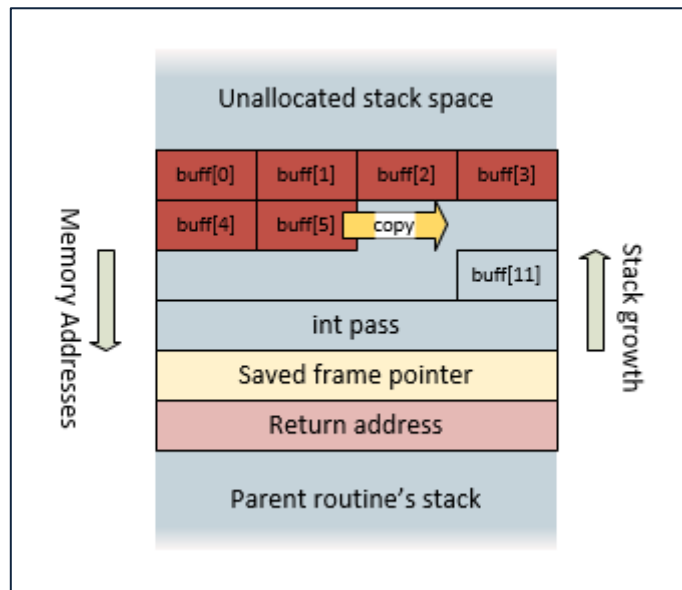


Figure 15: stack manipulation when writing data in the buffer

In the case more than 15 bytes of data are copied, then the `pass` variable starts to be overridden by the data copied, like in the figure below:

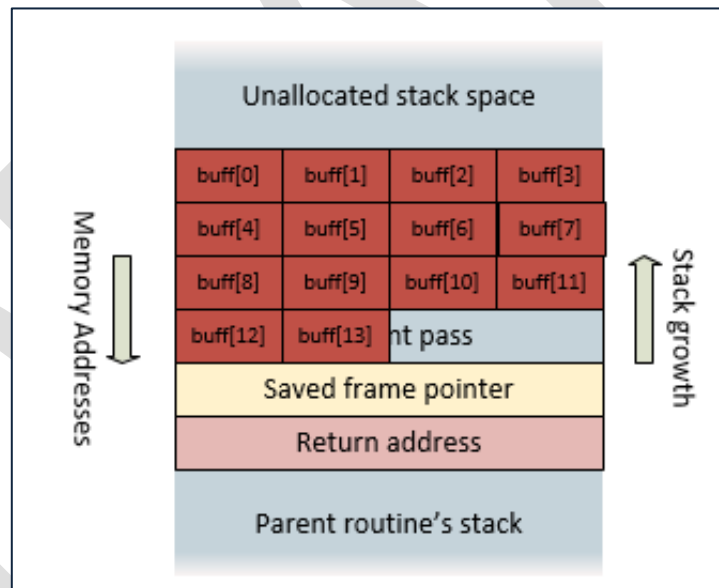


Figure 16: pass variable overwrite

So, in our case, if more than 15 bytes are copied, and data copied is not null, then our `pass` variable is no more equal to zero. Finally, the buffer overflow here lets a malicious user to access the privileged area without knowing the password:


```
11:34:30 [nzo:~/vessedia_tests] 139 $ ./stack1

Enter the password :
AAAAAAAAAAAAAAAAAAAA

Wrong Password

Root privileges given to the user
11:35:05 [nzo:~/vessedia_tests] $
```

Figure 17: privilege escalation without the correct password

Buffer overflow can also occur on heap and BSS. Here is an example of a heap-based buffer overflow:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX 32
6 #define LENGTH_USERID 8
7
8 void hash(char* test,char* test2){
9     return; //should derives code to a hash into userID
10}
11
12 int main(int argc, char *argv[]){
13     char *code; //Final returned code
14     char *userID;
15     char* key="thisisasupersecretpassword";
16
17     if(argc < 2)
18     {
19         printf("./%s <your code>",argv[0]);
20         return 1;
21     }
22
23     code = (char *)malloc(sizeof(char)*MAX);
24     userID = (char *)malloc(sizeof(char)*LENGTH_USERID);
25     memset(code,1,MAX);
26     memset(userID,0,LENGTH_USERID);
27
28     strcpy(userID,"guest");
29     strcpy(code,argv[1]); //vulnerability here
30
31     if(strncmp(key,code,26)==0){
32         printf("Your code is OK !!\n");
33         hash(code,userID);
34     }else{
35         printf("Bad code...\n");
36     }
37
38     if(strcmp(userID,"guest")==0){
39         printf("Welcome guest...\n");
40     }
41     else {
42         printf("Welcome registered user : %s !\n",userID);
43     }
44     free(code);
45     free(userID);
46
47     return 0;
48}
```

In this program, once again the authentication of a user is realized. Basically, one user is determined by a `userID` and a password. The idea of this code is to authenticate a user given its password. Once its password is entered, a hashing function return the correct user associated, and rights are given based on the `userID`.

Unfortunately, the program is subject to a heap-based overflow, permitting to overwrite the `userID`: two consequent allocations are made on the heap, at lines 23 and 24. That means that `code` and `userID` are actually following each other on the heap, and that when there is a buffer overflow in `code`, then `userID` may be overwritten. Especially here, we see that `code` is allocated a given size and that command line parameter is actually copied into it. However, this parameter can be of any size, and `strcpy` makes the copy byte by byte until the provided string to copy has a null-byte. Thus, `userID` may be overridden, and an access to a privileged area can be granted here:

```
11:49:17 [nzo:~/vessedia_tests] $ ./heap AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAroot
Bad code...
Welcome registered user : root !
11:49:19 [nzo:~/vessedia_tests] $
```

Figure 18 : heap overflow exploitation from the previous program

Finally, here is a buffer overflow occurring on BSS. Those buffer overflows are in general less exploitable because there is a need of some special configuration. Indeed, as already explained within the previous chapter, BSS is a memory region allocated before the heap, of a given size, because every global variable size going there is known in advance. Then, most of the variables that will be accessed are either stack or heap ones, and as it is not possible to write memory past the heap base (without arbitrary write), variables in BSS will most of the time not be affected by an overflow. However, here is a case of an exploitable BSS buffer overflow that may lead to execution flow hijacking:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char username[512] = {1};
5 void (*_atexit)(int) = exit;
6
7 void cp_username(char *name, const char *arg)
8 {
9     while((*name++ = *(arg++)));
10    *name = 0;
11}
12
13 int main(int argc, char **argv)
14{
15    if(argc != 2)
16        {
17        printf("[-] Usage : %s <username>\n", argv[0]);
18        exit(0);
19        }
20
21    cp_username(username, argv[1]);
22    printf("[+] Running program with username : %s\n", username);
23
24    _atexit(0);
25    return 0;
26}
```

Basically, this program simply asks for a user's name to the user, copy it to `username` variable, and finally executes the rest of its instructions based on the username (more than simply exit, but here it is a projection).

As we can see, there is a buffer allocated on the BSS segment, as well as a function pointer. The function `cp_username` does not check the length of the provided name to be copied, and as


```

3 #include <string.h>
4 #include <stdlib.h>
5
6 //[...]
7
8 typedef struct {
9     unsigned int address;
10    unsigned char size;
11    char* data;
12}packet_t;
13
14 void create_packet(raw_data_t* raw_data, packet_t* new_packet){
15     new_packet->address = (unsigned int)strtoul(raw_data, NULL, 16);
16     new_packet->size = (unsigned char)(raw_data[4]);
17     if (new_packet->size > 5 && new_packet->size < 255){
18         new_packet->data = malloc(new_packet->size*sizeof(char)+1);
19         memset(new_packet->data,0,new_packet->size+1);
20         memmove(new_packet->data,(char*)(raw_data+5),new_packet->size);
21     }
22     else{
23         new_packet->data=0;
24     }
25}
26
27 void debug_packet(packet_t* packet){
28     printf("contents of packet address is %i\n", packet->address);
29     printf("contents of packet size is %i\n", packet->size);
30     printf("contents of packet data[0] is %c\n", (packet->data)[0]);
31}
32
33//[...]
34
35 void main(){
36     raw_data_t* raw_data;
37     //[...]
38     while(server_on){
39         packet_t packet;
40         gather_raw_data(raw_data);
41         create_packet(raw_data,&packet);
42
43#ifdef DEBUG
44     debug_packet(&packet);
45#endif
46
47     clean_packet_data(&packet);
48     //[...]
49     }
50     printf("quitting server\n");
51}

```

The code basically does the following: a server is started, which is an infinite loop where each packet is gathered (we can imagine those packets are stored in a circular buffer by the network card) by the `gather_raw_data` function. As such, a packet structure is filled, which could be used for treatment by the `handle_packet` function. Finally, memory allocated is cleaned in `clean_packet_data` function. In the middle of the main function, one can see that in the case the program is compiled with the `DEBUG` parameter, then a debug function prints the contents of packets structures created that way.

As we can see in the declaration of the type `packet_t` above, a packet is formed by an address, which is a field of 4 byte, then a size on one byte, and finally its data, which can be of any size. As such, a packet is at least normally 5 bytes long, and the field size should normally indicate a

size of more than 5. So, what happens at line 17 is the fact that if the size indicated by an incoming packet is less than five, then the packet is invalid, and its data is set to zero.

But, when the packet is accessed in the debug function, then data is printed. As such, the pointer to packet data is dereferenced. When the size is less than five, typically in the case an attacker sends a packet with a size of zero, then the packet data field set to zero is dereferenced, leading to the vulnerability. That results in a segmentation fault, a kill of our program:

```
contents of packet address is 4660
contents of packet size is 0
contents of packet data is (null)
Erreur de segmentation
16:47:15 [nzo:~/vessedia tests] 139
```

Figure 21: segmentation fault due to null pointer dereference

3.2.3 Uninitialized variable utilization

C language does not provide a way to initialize automatically variables. As such, if a variable is declared without initialization, the content of this variable is actually undefined, as it will point to the memory previously in place (basically, stack space is allocated as is, without any push of an initializer variable on the space allocated). Variables may have some completely unexpected values, what may lead to security error, as in this variant of our authentication program:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      int pass;
7      char buff[16];
8
9      printf("\n Enter the password : \n");
10     fgets(buff, sizeof(buff), stdin);
11
12     if(strncmp(buff, "thegeekstuff", 16))
13     {
14         printf ("\n Wrong Password \n");
15     }
16     else
17     {
18         printf ("\n Correct Password \n");
19         pass = 1;
20     }
21
22     if(pass)
23     {
24         /* Now Give root or admin rights to user*/
25         printf ("\n Root privileges given to the user \n");
26     }
27
28     return 0;
29 }
```

Here, the `pass` variable is not initialized and will contains the value of the memory that was present when assigning the variable. So if the memory was not nullified, then the `pass` variable will not be equal to zero, which means that the user will be authenticated even without giving the correct password:

```
16:47:15 [nzo:~/vessedia_tests] 139 $ ./uninit
Enter the password :
vessedia

Wrong Password

Root privileges given to the user
16:52:15 [nzo:~/vessedia_tests] $
```

Figure 22: uninitialized variable utilisation

3.2.4 Double free

A double free vulnerability occurs when a dynamically allocated variable is freed twice. As a result, the heap management for the program becomes corrupted and the program has an undefined behavior. In some cases, a denial of service will occur while sometimes it may be possible for an attacker to alter the execution flow of the program.

This vulnerability really depends on how the heap management of the program is realized. So, in the case of the following snippet of code:

```
a = malloc(10);
b = malloc(10);

free(a);
free(a); // Double Free

c = malloc(10);
d = malloc(10);
```

When `a` is freed for the first time, the two lists of chunks become:

```
Allocated: head -> b -> tail
Freed: head -> a -> tail
```

When `a` is freed a second time, the lists become:

```
Allocated: head -> b -> tail
Freed: head -> a -> a -> tail
```

Then, when `c` and `d` are allocated, as they have the same size than `a`, they will both be allocated given the two free chunks pointed to `a`:

```
Allocated: head -> b -> c (points to address a) -> d (points to address a) -> tail
Freed: head -> tail
```

So, in the rest of the program, when `c` and `d` are manipulated, they both manipulates the same region of memory.

To illustrate an example of such a problem (in fact, this vulnerability is often only useful for exploitation of control flow in case of just-in-time-compiled code), let's take the following code that simulates a tiny Missiles platform management application. Basically, this platform is operated by a soldier, who can be called by generals to realize several operations:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define NB_USER 32
6
7 typedef struct{
8     int ID;
```

```

9     char password[10];
10    }user_t;
11
12    //[...]
13
14    void retrieve_password(user_t** user_list, int ID){
15        if (user_list[ID])
16            printf("\thi, here is your password : %s\n",user_list[ID]-
>password);
17        else
18            printf("\tnon existing user\n");
19    }
20
21    void create_user(user_t** user_list){
22        user_t* new_user = malloc(sizeof(user_t));
23        int i;
24        for(i=0;i<NB_USER;i++){
25            if(!user_list[i]){
26                new_user->ID = i;
27                printf("\tplease enter password (9 chars max) : ");
28                char tmp_buffer[10];
29                scanf("%s",tmp_buffer);
30                memmove(new_user->password,tmp_buffer,9);
31                user_list[i]=new_user;
32                printf("\tuser created, here is the ID : %i\n",i);
33                return;
34            }
35        }
36        printf("\tcan't create a new user!\n");
37        return;
38    }
39
40    void get_user_id(int* ID){
41        int i;
42        printf("\twhat's the ID to be used here ? : ");
43        scanf("%d",&i);
44        *ID=i;
45    }
46
47    void delete_user(user_t** user_list,int ID){
48        free(user_list[ID]);
49    }
50
51    //[...]
52
53    int main(int argc, const char * argv){
54        int loopout = 0;
55        int choice;
56        int current_ID=0;
57        user_t** user_list = malloc(NB_USER*sizeof(user_t*));
58        while (!loopout)
59            {
60                printf("\nMissiles platform management\n");
61                printf("please select:\n");
62                printf("1. create a new user\n");
63                printf("2. delete a user\n");
64                printf("3. retrieve user password\n");
65                printf("4. quit\n\tnum: ");
66                scanf("%i",&choice);
67                switch (choice)
68                    {
69                        case 1:
70                            create_user(user_list);

```

```

71         break;
72     case 2:
73         get_user_id(&current_ID);
74         delete_user(user_list,current_ID);
75         break;
76     case 3:
77         get_user_id(&current_ID);
78         retrieve_password(user_list,current_ID);
79         break;
80     case 4:
81         exit(0);
82         break;
83     default:
84         printf("wrong choice\n");
85         break;
86     }
87 }
88
89     return 0;
90 }

```

This code presents a menu for the operator to realize different operations of managing the users of the missile platform. When a colonel calls the operator, then the latter realizes all operations asked for because of his grade. However, multiple vulnerabilities can be found in this code because there are no sanitizing checks while manipulating the users. Indeed, those are manipulated through their ID, and operations can then be realized in whatever order. It is possible to delete the same user twice, or to ask for the password of a deleted user.

Let's imagine the following scenario (completely fancy to be honest):

First, the attacker infiltrates the army, and becomes Skywalker officer. He then imitates colonel Vador and calls the operator of the missile platform: “– Hi administrator, here is colonel Vador. I need you to delete my account because it may have been compromised.”


```

10:32:52 [nzo:~/vessedia_tests] $ ./user_missiles
Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 1
    please enter password (9 chars max) : vador123
    user created, here is the ID : 0

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 2
    what's the ID to be used here ? : 0

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 2
    what's the ID to be used here ? : 0

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 1
    please enter password (9 chars max) : skywalker
    user created, here is the ID : 1

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 1
    please enter password (9 chars max) : vador123
    user created, here is the ID : 2

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 3
    what's the ID to be used here ? : 1
    hi there, here is your password : vador123

Missiles platform management
please select:

```

identity theft

The administrator executes that, and then receives a second call, also from the same attacker: “Hi administrator, here is colonel Palpatine. I need you to delete colonel Vador’s account, and then create him a new account. Then, you should also create a new account for the new officer Skywalker.”

The administrator executes the order, calls officer Skywalker, which is the attacker, to create his account, and finally calls colonel Vador, to create his account. Once all of this has been realized, the attacker calls to request his password because “- it does not work”.

In this case, colonel Vador’s account has been deleted twice, its user has been freed twice. Then, the attacker gets an account which points to the old memory of colonel Vador’s account (because a user takes the same size in memory at whatever times), and colonel Vador gets also a new account, pointing also to his own old memory area. As such, colonel Vador’s new account and Officer Skywalker’s account point to the same memory. As colonel Vador’s account is created last, his password is put in that memory region. Then, when the attacker asks for his password, he indeed retrieves colonel Vador’s one. As such, the attacker gained colonel Vador’s privileges on the missiles platform.

Here is a screenshot illustrating that scenario:

Figure 23: double free vulnerability leveraged into an

3.2.5 Use-after-free

Use-after-free vulnerability is, as indicated by his name, caused by the use of an already `freed` dangling pointer. Once again, as in the previous vulnerability, the heap memory manager mechanisms are important to take advantage of this vulnerability.

For example, in the case a chunk is freed, it can be used afterwards by the memory manager for another object. When the use of the freed object occurs, the program believes to manage another object than the one really managed.

The Missile management platform used in the previous paragraph is also vulnerable to use-after-free. Let's get back to the scenario of an attacker calling the operator: "Hi operator, here is colonel Palpatine. Please delete my account". The attacker then waits few days, a user has been created. As the user created take the same amount of space than Colonel Palpatine's user identity, the memory chunk for this new user is the same than the one used for colonel Palpatine. The attacker can call again the operator and say "-Hi there, here is colonel Palpatine and here is my ID. Please tell me my password back". The operator will give the password that points now to the one of the newly user created. The attacker gains another credential to use on the missile platform.

Here is a screenshot to illustrate this vulnerability:

```
11:22:56 + [nzo:~/vessedia_tests] 130 $ ./user_missiles
Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 1
    please enter password (9 chars max) : palpatine
    user created, here is the ID : 0

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 2
    what's the ID to be used here ? : 0

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 1
    please enter password (9 chars max) : toto1234
    user created, here is the ID : 1

Missiles platform management
please select:
1. create a new user
2. delete a user
3. retrieve user password
4. quit
    num: 3
    what's the ID to be used here ? : 0
    hi there, here is your password : toto1234
```

Figure 24: use-after-free vulnerability leveraged into identity theft

3.2.6 Integer Overflow

In C, every type data is coded on a fixed size bit field. For example, an integer as defined with the `int` keyword is coded on 32 bits even on 64 bits platforms. As such, an integer can be represented by 31 bits for its value, and one bit for its sign. An `int` must have a value between $-2\,147\,483\,648$ to $2\,147\,483\,647$.

When an operation is realized on such a data field, and this operation creates a data field that is out-of-bound of the theoretical limits, the result is in fact computed using the modulo arithmetic fusing the size of the field. This often results in miscalculations, and that can lead to out-of-bounds write. For example, flight 501 of the rocket Ariane 5 crashed because of such a vulnerability.

To illustrate the vulnerability, let's take a modified version of our IOT server from the paragraph 3.2.2 :

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  typedef struct {
6      unsigned int address;
7      unsigned char size;
8      char* data;
9  }packet;
10
11 void handle_packet(char* raw_data, packet* new_packet){
12     unsigned char new_size;
13     new_packet->address = (unsigned int)strtoul(raw_data, NULL, 16);
14     new_packet->size = (unsigned char) (raw_data[4]);
15     if (new_packet->size > 5){
16         new_size = new_packet->size+1;
17         new_packet->data = malloc(new_size);
18         memset(new_packet->data,0,new_size);
19         memmove(new_packet->data, (char*) (raw_data+5),new_packet->size);
20         raw_data=raw_data+new_packet->size-5;
21     }
22     else{
23         new_packet->data=malloc(sizeof(char));
24         memset(new_packet->data,0,1);
25         raw_data=raw_data+5;
26     }
27 }
28
29 void clean_packet_data(packet* packet){
30     free(packet->data);
31 }
32
33 //[...]
34 void main(){
35     char* raw_data;
36     init_raw_data(raw_data);
37     //[...]
38     int server_on =1;
39     while(server_on){
40         packet packet;
41         gather_raw_data(raw_data);
42         handle_packet(raw_data,&packet);
43         clean_packet_data(&packet);
44         server_on=0;
45     }
46     printf("quitting server\n");
47 }
```

In this version, the null pointer dereference vulnerability has been removed, by making sure that packet data is allocated at least even if the size is less than five. However, the check on line 14 has been modified and it is no more checked that the size is less than 255.

The size is an unsigned char, meaning the data is coded on 8 bits, without any bit for the sign. The size can have a value between 0 to 255. As such, if a packet with a size of 255 is read, then the `new_size` variable, which also an unsigned char, is set to $(255+1) \% 2^8 = 0$. As such, when the `memmove` operation is called afterwards, an out-of-bounds write occurs. There is also to note that the `malloc` return code is not checked, that can lead also to an out-of-bounds write.

3.2.7 Off-by-one

Off-by-one bugs are basically buffer overflows whose outreach is a writing of only one more byte. They got special subclass vulnerabilities as by overwriting only one byte, exploitation methods are by far different than those in classical buffer overflows. Especially, those are much more difficult to exploit, and the only meaningful cases for an exploitation is when a given function pointer can be overwritten by one byte. Moreover, detecting them is also not trivial and requires a deep understanding of C standard library.

To illustrate this class of vulnerabilities, let's take again the authorization handler from the buffer overflow case. Here, the developer modified the program in an attempt to correct the buffer overflow, and to have a better amplitude on the input, while retaining the fact that the password is still coded on 15 bytes:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      int not_logged = 1;
7      char buff[15];
8      char s[64];
9
10     memset(buff, 0, sizeof(buff));
11
12     printf("\n Enter the password : \n");
13     fgets(s,63,stdin);
14     strncat(buff, s, sizeof(buff));
15
16     if(strcmp(buff, "thegeekstuff"))
17     {
18         printf ("\n Wrong Password \n");
19     }
20     else
21     {
22         printf ("\n Correct Password \n");
23         not_logged = 0;
24     }
25
26     if (!not_logged)
27     {
28         /* Now Give root or admin rights to user*/
29         printf ("\n Root privileges given to the user \n");
30     }
31
32     return 0;
33 }
```

In this example, the bug comes from an improper call to `strncat(3)` function. As it is said in the manual page, this function adds a terminating null byte. So, as the destination buffer size doesn't take this into account, an overflow of 1 byte might happen, overwriting stack data, which is here the `not_logged` variable. As such, a user with a wrong password can still access the privileged area:

```
17:01:32 [nzo:~/vessedia_tests] $ ./offbyone
Enter the password :
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Wrong Password
Root privileges given to the user
17:01:41 [nzo:~/vessedia_tests] $
```

Figure 25: execution of the off-by-one program

3.2.8 Format String

Format string vulnerability is a mishandling of the `printf`-like functions parameters, and is one of the most dangerous vulnerability because it gives attacker the ability to read or write anywhere in memory. To understand this vulnerability, one has to understand how `printf`-like functions work. In normal cases, the `printf` functions use a formatter, which is a string indicating the format of the variables to print, and then the list of all variables to print. For example, one call to `printf` might be the following:

```
printf("hello %s",username);      where "hello %s" is the formatter.
```

When such a function is called, its parameters are passed on the stack as already explained in paragraph 3.1. Our previous call has the following stack:

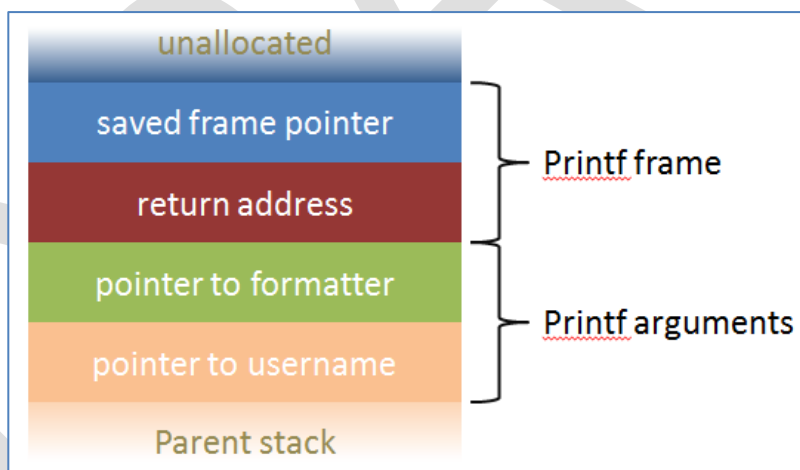


Figure 26: stack layout when calling printf

Then, internally and in a simplified version, the `printf` function parses its first argument, and stops at each identifier like `%s`. When it encounters such an identifier, the next argument present on the stack is gathered to form the final string to print. Finally, the final string is sent to `stdout`.

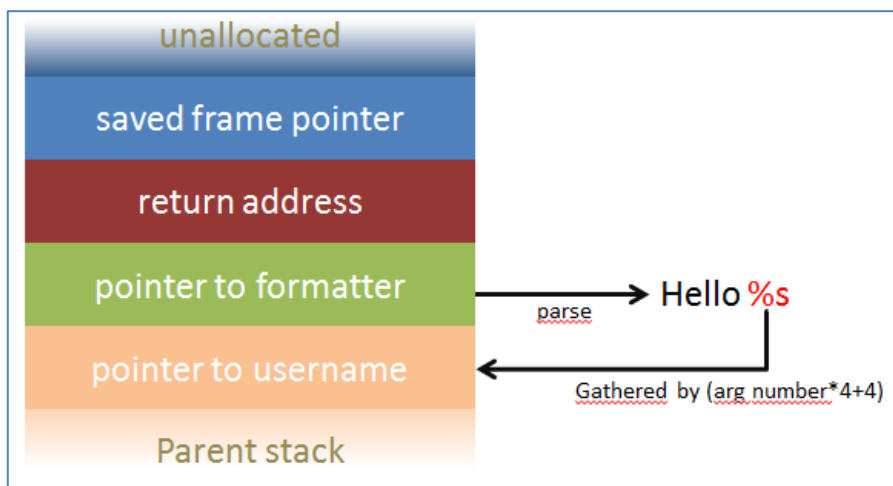


Figure 27: internal working of printf

There is a need to have the same number of variables than the number of identifiers in the formatter, because in this case there is the right number of arguments to pop off the stack to be able to form the final string.

A format string vulnerability occurs in reality when this condition is not verified. In the case for example there is no variable as a second argument, while a formatter with one identifier is passed as the first argument, then printf will still try to read the stack for its second argument. As this second argument is not present on stack, then the previous value of memory at this place will be read instead and added to the final string. So, for example, a "%x" identifier passed as the first argument, without a variable as a second argument, will print the first value on the stack in hexadecimal, leading to an arbitrary read condition. To illustrate this, let's take again the authorization management application. This time, the developer chose to use a configuration file to have the root password:

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <string.h>
4
5 int main(int argc, char *argv[]){
6     int not_logged=1;
7     char symbol;
8     FILE *secret_file = fopen("config.txt", "r");
9     char buffer[128];
10    char secret[32]={0};
11
12    if(secret_file != NULL)
13    {
14        size_t newLen = fread(secret, sizeof(char), 31, secret_file);
15        if ( ferror( secret_file ) != 0 )
16        {
17            fputs("Error reading file", stderr);
18        } else
19        {
20            secret[newLen++] = '\0'; /* Just to be safe. */
21        }
22        fclose(secret_file);
23    }
24
25    printf("\n Enter the password : \n");
26    fgets(buffer, sizeof(buffer), stdin);
27
28    if(strcmp(buffer, secret))
29    {
30        printf ("\n Wrong Password, you have entered : ");
31        printf(buffer);

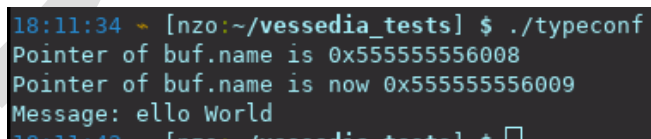
```


A type confusion vulnerability occurs when a variable can point to two different types, and it is treated as of the wrong type. It is especially true while using the union keyword in C. Unfortunately, this vulnerability is not common in C, and is much more useful in object language, because in those, methods are parts of the object. It is possible to call wrong methods on objects which can lead to a control flow hijack. Here is an example in C, without any security issues after exploitation:

```
1 #define NAME_TYPE 1
2 #define ID_TYPE 2
3
4 struct MessageBuffer
5 {
6     int msgType;
7     union {
8         char *name;
9         int nameID;
10    };
11};
12
13 int main (int argc, char **argv) {
14     struct MessageBuffer buf;
15     char *defaultMessage = "Hello World";
16
17     buf.msgType = NAME_TYPE;
18     buf.name = defaultMessage;
19     printf("Pointer of buf.name is %p\n", buf.name);
20     /* This particular value for nameID is used to make the code
21     architecture-independent. If coming from untrusted input, it could be any
22     value. */
23     buf.nameID = (int)(defaultMessage + 1);
24     printf("Pointer of buf.name is now %p\n", buf.name);
25     if (buf.msgType == NAME_TYPE) {
26         printf("Message: %s\n", buf.name);
27     }
28     else {
29         printf("Message: Use ID %d\n", buf.nameID);
30     }
31 }
```

The code intends to process the message as a `NAME_TYPE`, and sets the default message to "Hello World." However, since both `buf.name` and `buf.nameID` are part of the same union, they can act as aliases for the same memory location, depending on memory layout after compilation. As a result, modification of `buf.nameID` (which is an `int`) can effectively modify the pointer that is stored in `buf.name`, a string.

The execution of the program generates output such as:



```
18:11:34 ~ [nzo:~/vessedia_tests] $ ./typeconf
Pointer of buf.name is 0x555555556008
Pointer of buf.name is now 0x555555556009
Message: ello World
18:11:42 ~ [nzo:~/vessedia_tests] $
```

Figure 30: example of type confusion

The pointer `buf.name` was changed, even though `buf.name` was not explicitly modified.

3.3 Cryptographic vulnerabilities

This paragraph won't go into many details, as explaining cryptographic background there is a hard and tedious task that requires another report on its own.

3.3.1 *Non-respect to cryptographic standards*

Although those are not direct vulnerabilities, there is today a defined standard on most of the cryptographic functions used. Indeed, using algorithms that have been cracked, or using algorithms that are known to be vulnerable to a theoretical attack, may lead a user to create a cryptographic attack on your application. This could especially be used to gather secrets held by the application.

For example, in Europe, the SOG-IS agreement provides a baseline indicating approved cryptographic algorithms⁸.

More than algorithms, there is also a need for any algorithm to verify some sets of cryptographic properties. Especially, those sets of properties are important because they are ensuring theoretical non-exploitability. One example of it is what is called the distribution on a hashing algorithm, ensuring that if one byte of the original message is changed, then at least half of the total bytes of the hashed message is changed.

In most cases, it is verified that the algorithm was not self-developed and got no cryptographic review.

3.3.2 *Misuse of cryptographic algorithms*

Cryptography is a hard topic. As such, an evaluator can often see a misuse of cryptographic algorithms. Indeed, the developer tried to obtain a set of cryptographic properties, but the algorithm chosen may not bring all those properties. For example, two of cryptographic properties are authentication, which is the act of confirming the truth of an attribute of a single piece of data claimed true by an entity, like its identity, and integrity, assuring the accuracy and completeness of some data over its entire lifecycle. Most hashing algorithms provide only the latter, while an asymmetric key algorithm like RSA can provide both. However, sometimes, hashing is used to authenticate one user on an application.

3.4 C vulnerabilities depending on the environment

3.4.1 *Race condition*

A race condition occurs when there is some time lapse between the access to a resource, and then the use or destruction of this resource. If this resource can be controlled by an attacker, then he may manipulate it in this meantime. The name comes from the fact that the attacker is in race with the legit program to modify the accessed resource in between. This vulnerability can be used to elevate one's privileges.

As an example, let's say a program has to authenticate users against a credential given in a ciphered file. To do so, the program first opens the ciphered file to decipher it into a temporary file on the system (that does not have any system's controlled access), asks the user about its credentials, then read the deciphered file to compare the credential given by the user, and finally destroys the temporary file:

```
1 #include <stdio.h>
```

⁸ https://www.sogis.org/uk/supporting_doc_en.html

```

2  #include <unistd.h>
3  #include <string.h>
4
5  void decipher_file(char* secret_file,char* temp_file, FILE**
temp_file_handle){
6      //this function basically opens the secret file and the temp file
7      // then deciphers the contents of secret file into temp file
8      //and finally returns a handle towards temp_file
9  }
10
11 int main(int argc, char *argv[]){
12     int not_logged=1;
13     char symbol;
14     char secret_file[]="config.txt";
15     char temp_file[]="/tmp/tmp_deciphered.txt";
16     FILE* temp_file_handle;
17     char buffer[32]={0};
18     char secret[32]={0};
19
20     printf("deciphering file\n");
21     decipher_file(secret_file,temp_file,&temp_file_handle);
22     printf("\n Enter the password : \n");
23     fgets(buffer, sizeof(buffer), stdin);
24     if(temp_file_handle != NULL)
25     {
26         size_t newLen = fread(secret, sizeof(char), 32,
temp_file_handle);
27         if ( ferror( temp_file_handle ) != 0 ) {
28             fputs("Error reading file", stderr);
29         } else {
30             secret[newLen++] = '\0'; /* Just to be safe. */
31         }
32         fclose(temp_file_handle);
33     }
34
35     if(strcmp(buffer, secret))
36     {
37         printf ("\n Wrong Password\n");
38     }
39     else
40     {
41         printf ("\n Correct Password \n");
42         not_logged = 0;
43     }
44
45     if (!not_logged)
46     {
47         printf ("\n Root privileges given to the user \n");
48     }
49
50     return 0;
51 }

```

Of course, if an attacker is able to modify the file while the user is entering the password, because a handle is already opened on it, then he can control what will be the value the credentials of the user will be tested against.

Here is a screenshot illustrating this case (please make sure to look at the times). First, a run of the program is realized, the user does not have the correct credentials. Then the same run is realized, but while the user is entering the password, the `tmp_file` is modified. He can then authenticate himself:

```

18:28:36 ~ [nzo:~] $
18:28:36 ~ [nzo:~] $ cat /tmp/tmp_deciphered.txt
secretpassword
18:29:15 ~ [nzo:~] $ echo "abcd" > /tmp/tmp_deciphered.txt
18:29:24 ~ [nzo:~] $

18:28:49 ~ [nzo:~/vessedia_tests] $ ./racecondition
deciphering file

Enter the password :
abcd

Wrong Password
18:29:06 ~ [nzo:~/vessedia_tests] $ ./racecondition
deciphering file

Enter the password :
abcd

Correct Password

Root privileges given to the user
18:29:27 ~ [nzo:~/vessedia_tests] $

```

Figure 31: race condition illustration

3.4.2 Path manipulation

Path Manipulation vulnerabilities occur when a program attempts to access an external resource, that may be controlled by the attacker, without controlling that the path given to this external resource is correct. As such, the path may point to an invalid resource.

For example, let's say a program run by a privileged user invoke the `ls` command to check about some files on the system:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main( int argc, char *argv[] )
5  {
6      FILE *fp;
7      char path[1035];
8
9      /* Open the command for reading. */
10     fp = popen("ls /etc/", "r");
11     if (fp == NULL)
12     {
13         printf("Failed to run command\n" );
14         exit(1);
15     }
16     while (fgets(path, sizeof(path)-1, fp) != NULL)
17     {
18         //do something
19     }
20
21     /* close */
22     pclose(fp);
23
24     return 0;
25 }

```

One can see at line 10, that a call to the `ls` command is realized, without using an absolute path. In such a scenario, the underlying operating system will first search the `ls` command in the file system, and then invoke it. On Linux, that search is done in the order given by the `PATH` environment variable, which is a series of file system paths interspersed with the delimiter `:`. The

search is done as this: the first path in `PATH` is looked into for the presence of a `ls` binary. If the `ls` binary is found, then it is invoked, meaning the command is invoked. Otherwise, the second path is taken and the same examination is realized. This continues until the `ls` binary is found in one of the underlying path, or until there is no more path to investigate given by `PATH` variable.

So, if the attacker manipulates the `PATH` variable by setting it to `/tmp` and then creates a false `ls` binary in this `/tmp` folder, then this attacker-controlled program will be called instead. Here is an example to illustrate that kind of manipulation from the program above:

```
21:32:57 ~ [nzo:~/vessedia_tests/path_manipulation] $ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
21:33:00 ~ [nzo:~/vessedia_tests/path_manipulation] $ ./path
path
path.c
21:33:03 ~ [nzo:~/vessedia_tests/path_manipulation] $ cat /tmp/ls
#!/bin/bash
echo "you're hacked"
21:33:23 ~ [nzo:~/vessedia_tests/path_manipulation] $ PATH=/tmp
21:33:28 ~ [nzo:~/vessedia_tests/path_manipulation] $ ./path
you're hacked
21:33:31 ~ [nzo:~/vessedia_tests/path_manipulation] $
```

Figure 32: path manipulation example

3.4.3 SQL Injection

A SQL injection appears when a program runs some SQL queries, without separating the request from the fields of the query. That means that the fields may be manipulated by an attacker to modify the meaning of the request. Let's take for the example the following program that authenticates a user using the MySQL API :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mysql.h>
4
5  #define QUERY_LEN 512
6
7  int main( int argc, char *argv[] )
8  {
9      char name[32];
10     char password[32];
11
12     MYSQL mysql;
13     mysql_init(&mysql);
14     mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"option");
15
16     if(mysql_real_connect(&mysql,"www.goldzoneweb.info","mon_pseudo","****
17     *","ma_base",0,NULL,0))
18     {
19         printf("\n Enter your name : \n");
20         fgets(name, sizeof(name), stdin);
21
22         printf("\n Enter your password : \n");
23         fgets(password, sizeof(password), stdin);
24
25         char myquery[QUERY_LEN];
26         sprintf(myquery, "select * from users where name='%s' and
27         password='%s'", name, password);
28
29         if (mysql_query(conn, myquery ))
30         {
```

```

28         fprintf(stderr, "%s\n", mysql_error(conn));
29         exit(1);
30     }
31
32     MYSQL_RES *result = NULL;
33     MYSQL_ROW row;
34     result = mysql_use_result(&mysql);
35     if(result != NULL)
36     {
37         printf("Welcome authenticated user : %s", name);
38     }
39     mysql_free_result(result);
40     mysql_close(&mysql);
41 }
42 else
43 {
44     printf("an error occured\n");
45 }
46
47     return 0;
48 }
49

```

As one can see, the program uses the `sprintf` function in order to concatenate the query with the fields given by the user. If the user uses the following credentials `admin/adminpass`, then the query becomes:

```
select * from users where name='admin' and password='adminpass'
```

It is then checked if the result of the query is not null. In this case, that means the user exists.

An attacker can leverage the vulnerability by providing the following credentials to the application: `admin' -- /pass`. As such, the query becomes the following:

```
select * from users where name ='admin' -- and password='pass'
```

The end of the query is treated as a comment and the query becomes:

```
select * from users where name ='admin'
```

So, only the verification on name will be conducted. The attacker can finally gain privileged access.

3.4.4 Command Injection

Command injection vulnerability occurs when a program makes a call to an external command with parameters provided by the attacker. As such, the attacker can potentially make additional calls to other commands. This call to other commands can really be useful if the program is run with other privileges than the user one. Indeed, the injected commands will be executed in the context of the program privileges. Let's take for example the following program, which is an ergonomic helper on using SSH:

```

1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <string.h>
4
5     void connect_to(){
6         char serv_name[32];
7         char username[32];
8         char pass[32];
9         FILE *fp;
10        char path[1024];

```

```

11
12     printf("\n Enter server name : ");
13     fgets(serv_name, sizeof(serv_name), stdin);
14     serv_name[strcspn(serv_name, "\n")] = 0;
15
16     printf("\n Enter your user name : ");
17     fgets(username, sizeof(username), stdin);
18     username[strcspn(username, "\n")] = 0;
19
20     printf("\n Enter your password : ");
21     fgets(pass, sizeof(pass), stdin);
22     pass[strcspn(pass, "\n")] = 0;
23
24     char command[512];
25     sprintf(command, "sshpass -p %s ssh %s@%s",
pass,username,serv_name);
26
27     fp = popen(command, "r");
28     if (fp == NULL) {
29         printf("Failed to run server\n" );
30         exit(1);
31     }
32
33     while (fgets(path, sizeof(path)-1, fp) != NULL) {
34         printf("%s", path);
35     }
36
37     pclose(fp);
38     //now, output must be check for authentication error, and then one
can use the ssh connection
39
40 }
41
42
43 int main(int argc, const char * argv[]){
44     int loopout = 0;
45     int choice;
46     char a;
47     while (!loopout)
48     {
49         printf("SSH Helper\n");
50         printf("please select:\n");
51         printf("1. connect to a SSH Server\n");
52         printf("2. retrieve/copy a file on a Server\n");
53         printf("3. managing SSH authentications\n");
54         printf("4. quit\n\tnum: ");
55         scanf("%i",&choice);
56         a = getchar();
57         switch (choice)
58         {
59             case 1:
60                 connect_to();
61                 break;
62             case 2:
63                 retrieve_or_copy();
64                 break;
65             case 3:
66                 manage_ssh_authent();
67                 break;
68             case 4:
69                 exit(0);
70                 break;
71             default:

```

```

72         printf("wrong choice\n");
73         break;
74     }
75 }
76 return 0;
77 }
78

```

We see this code provides a menu to the user, asking for what the user wants to do with SSH. One of the options lets the user to connect to a distant server, then to obtain an access to this distant shell. When this option is chosen, the user is asked for several parameters, that are then injected into a call to the SSH command. As such, an attacker may for example provide the following as a server name "server || cat /etc/passwd", thus the cat command is also executed in the context of the program.

Here is an illustration of this vulnerability. The current user, nzo, cannot read the file `secrets`, owned by `root` (1). Our SSH helper is however `setuid`, which means it is executed in the context of its owner, `root` (2). The attacker can leverage the example to read the `secrets` file:

```

20:53:46 ~ [nzo:~/vessedia_tests/command_injection] $ ll
total 24
-rw----- 1 root root    20 déc.  7 18:59 secrets
-rwsr-xr-x 1 root root 17256 déc.  7 19:27 SSHHelper
20:53:50 ~ [nzo:~/vessedia_tests/command_injection] $ cat secrets
cat: secrets: Permission non accordée
20:53:52 ~ [nzo:~/vessedia_tests/command_injection] 1 $ ./SSHHelper
SSH Helper
please select:
1. connect to a SSH Server
2. retrieve/copy a file on a Server
3. managing SSH authentications
4. quit
    num: 1

Enter server name : localhost||cat secrets

Enter your user name : tt

Enter your password : t
ssh: connect to host localhost port 22: Cannot assign requested address
this is secret data

SSH Helper
please select:
1. connect to a SSH Server
2. retrieve/copy a file on a Server
3. managing SSH authentications
4. quit
    num: ^C

```

Figure 33: elevating its privileges through command injection

3.4.5 Logic bugs

Logic bugs are the result of when the logic is not correctly calculated in a program, most often due to operators precedence, or the missing of some parenthesis. Let's take the following program, once again similar to the authentication program already presented:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  #define FAIL 0
5  #define SUCCESS 1
6
7  int AuthenticateUser(char *password) {

```

```

8         return !(strcmp(password,"mysuperpass"));
9     }
10
11     int main(void)
12     {
13         char buff[16];
14
15         printf("\n Enter the password : \n");
16         fgets(buff,sizeof(buff),stdin);
17
18         int isUser = FAIL;
19
20         if (isUser = AuthenticateUser(buff) == FAIL)
21         {
22             printf("\n Wrong password\n");
23         }
24         else
25         {
26             printf("\n Good password\n");
27             isUser = SUCCESS;
28         }
29
30         if(isUser == SUCCESS)
31         {
32             /* Now Give root or admin rights to user*/
33             printf ("\n Root privileges given to the user \n");
34         }
35
36         return 0;
37     }

```

Once again, the user is asked for its password, which is compared to a hardcoded one. In the case both values match, then the user is given privileges. Here, the variable holding the result of the authentication is put on the stack after the buff variable, which means that in any case, no buffer overflow can overwrite it. However, as one can see, the method that authenticates the user is called within an if statement (at line 30) with incorrect operator precedence logic. Because the comparison operator "==" has a higher precedence than the assignment operator "=", the comparison operator will be evaluated first and if the method returns FAIL then the comparison will be true, the return variable will be set to true and SUCCESS will be returned. As such, any password can provide the user privileges:

```

18:08:23 [nzo:~/vessedia_tests] $ ./logic_bug
Enter the password :
ABC
Wrong password
Root privileges given to the user
18:08:28 [nzo:~/vessedia_tests] $ ./logic_bug
Enter the password :
EFG
Wrong password
Root privileges given to the user
19:08:33 [nzo:~/vessedia_tests] $

```

Figure 34: logic bug leading to wrong privileges given to the user

3.4.6 Contextual vulnerabilities

A number of contextual vulnerabilities can arise. Those vulnerabilities are not bugs directly integrated in the software, but problems in a security functional component of the software, interacting with its environment. As those are not related to the code in itself, but are tightly linked to the context, here is a non-exhaustive list of named potential vulnerabilities that can be found:

- A wrong authentication of entities. Here, one can be authenticated in a wrong manner on the application, thus leading to privilege escalation and identity theft.
- An absence of protection of assets. Those assets could be for example, the ciphering keys used by encryption software, stored in clear text in a database. As such, the compromise of those assets leads to security problems.
- An absence of measures to prevent denials of service.
- An absence of logging utilities. This vulnerability is often overlooked, but in highly critical environment, logging is able to provide legal insurance as well as a way for an investigation to be held.
- The modification of the business parameters. One example here could be a financial program, using a configuration file. If the configuration file permits to alter taxes or rates, then the whole business can be impacted.

Chapter 4 On using Frama-C within the proposed methodology

This paragraph introduces how to incorporate Frama-C tools analysis within the proposed methodology of the paragraph 2.2. First, Frama-C will be briefly presented. In a second hand, a way to use concretely Frama-C is presented.

4.1 What is Frama-C?

4.1.1 Description

Frama-C is a static C/C++ source code analyzer, aimed at validation and proof of specifications of code. As such, it was proposed for highly critical development, to ensure the safety of the code afterwards. It is built on Ocaml, and has a plug-in architecture: a generic kernel centralizes information and conduct analyses, and plug-ins may communicate with the kernel and the other plug-ins via the kernel to provide analyses treatments and results.

The summed-up functioning of Frama-C is the following: First, the code is preprocessed by the gcc compiler. It produces a code with macro-expansion, header files inclusion and trigraph replacement. From this, Frama-C builds an abstract syntax tree (AST), i.e. a tree formed by the tokens of the code, obtained via the parsing of it. Finally, operations of source code analysis are directly computed on the AST (certainly while doing AST annotation). Finally, this AST and the results are operated by the several plugins to provide a deeper analysis.

The principal following plugins are shipped with the default Frama-C installation:

- Evolved value analysis (EVA): this plugin realizes a lot of operations based on abstract interpretation. The first ones are to be able to compute at any statement in the program the different values of the variables from the global and current scope. It also realizes several checks on memory accesses, like checking if memory accessed in tables or by pointers is valid. Integer arithmetics are also checked.
- E-ACSL plugin: this plugin provides the ability to process E-ACSL statements within the source-code. Those statements are in fact a language for proof verification, and formal properties are deduced from them. As such, those properties can then be verified while running other analyses.
- WP plugin: this plugin aids in assisting with proof verification. It will basically check any formal proof to be verified in the source code, helps with indications when the proof cannot be verified, and can run multiple provers on those proofs.
- Slicing plugin: this plugin produces a slicing of the code to generate an output still compiling, but with only the statements verifying certain properties.
- Impact plugin: this plugin outputs the statements that are impacted by the modification of a given statement. This plugin seems to rely on the slicing plugin to generates its results.
- Metrics plugin: this plugin provides several statistics about the code, like cyclomatic complexity and EVA coverage estimation.

A more thorough description is given within D3.3 document of VESSEDIA project.

4.1.2 *Frama-C's intended use*

Frama-C was intended for code safety, primarily required in highly critical domains like aeronautics, space and nuclear. As such, it follows to the following environment conditions:

- In general, code in such environments run on a single core and use very few system and user interaction. They are also in general quite small, and do not use libraries much.
- Those codes need to verify the enforced development process, and formal properties. So, whilst code is developed, formal proofs can also be developed.
- One of the biggest aims is to ensure there is no undefined nor unspecified behavior in the program, even in a critical state. As such, the program should be exempt of any bugs or run-time errors.
- The other big aim is to ensure the source code is compliant with some specifications, like MISRA-C⁹ for example.

Frama-C seems to be used in the following way in actual industrial environment:

- the main code is developed. In the meantime, specifications proofs and formal proofs are developed. Also, in order to be able to verify all proofs, E-ACSL specifications are developed.
- Once everything is developed, then Frama-C is used to ensure the code verifies the formal properties developed along with it, by means of proofs, as well as ensuring the absence of bugs for a code that is self-supporting. Especially, it can be used to ensure that variables are in a given range at a certain point of the program, and it can also make sure that all variables are initialized for example.
- In the case Frama-C tool discovers an error or a property that is not proven to be satisfied by the code, the code can be modified. Then, Frama-C is run again on the code, etc. until it says there is no error and all proofs are verified.

4.1.3 *A brief discussion on using Frama-C for security code review*

Formal specifications can only be produced by highly mature corporations, where skilled engineers are dedicated to this job. There is a huge probability that any code to be security reviewed is not shipped with any formal specifications, unless this code is to be used in highly critical environments.

Moreover, here are some vulnerabilities that can be detected by Frama-C:

- Buffer overflows,
- Double free,
- Null Dereference,
- Use-after-free,
- Integer Overflows,
- Uninitialized variables.

Frama-C appears so to be an interesting tool only for certain categories of code, where the following properties are met:

- code is deeply mastered by the developer (no use of frameworks nor a lot of additional libraries)
- code has low interaction with its environment and users
- code does not rely on implicit system treatment of it

⁹ <https://www.misra.org.uk/Publications/tabid/57/Default.aspx>

- code follows a specification to avoid potential flaws

Those codes can be found in embedded systems nowadays. Unfortunately, many of the new embedded systems (IOT and the like) are coded by less-mature organizations, where those categories of code are still not developed.

4.2 Integration of the modified Frama-C into the proposed methodology

4.2.1 Using Frama-C on the automated review part

4.2.1.1 Modifying Frama-C to support more codes

Frama-C appears to not be as efficient for codes that require a lot of interaction with their environment, because those interactions need modelling, and is not able to analyze the standard functions calls without added specifications.

On the latter, an analysis of Frama-C internals reveals that Frama-C defines what is called stubs for the standard Libc functions. Those stubs are actually E-ACSL annotations on top of every libc functions, ensuring some formal properties. Within those stubs, it appears that there is most of the time no annotations regarding potential security problems involved by the use of those functions. For example, here is the stub of the `fgets` function (which will be modified later on):

```
/*@
  requires valid_stream: \valid(stream);
  assigns s[0..size] \from indirect:size, indirect:*stream;
  assigns \result \from s, indirect:size, indirect:*stream;
  ensures result_null_or_same: \result == \null || \result == s;
  ensures terminated_string_on_success:
    \result != \null ==> valid_string(s);
*/
```

What this stub does is the following:

- By calling `\valid(stream)`, it is verified that `stream` is allocated memory
- Assigns clauses are here to assign symbolic values to the destination buffer
- Ensures clauses add additional properties to the symbolic values. It is for example added that the result is either the destination buffer assigned by the previous clauses or the null pointer.

As one can denote, there is no condition to make sure that the destination buffer size is enough to hold the stream. As such, a buffer overflow won't be detected there.

Those stubs can be partially adapted to security reviews. As a reminder, given that high constraints are often given to the reviewer, the reviewer should be able to use Frama-C quickly on the code to analyze, without having to study in-depth the code first. The idea is to ensure that Frama-C detects at least potential dangerous functions calls that may indicate to the reviewer where potential vulnerabilities in the source code reside.

To do so, E-ACSL functionalities have been studied a bit more in-depth. Basically, it is possible to ensure formal properties on a fixed sized memory area, but otherwise, it is not directly possible to model environment interaction in case the environment is not modeled first (e.g. it's not possible to write completely generic environment interactions) or to create on-the-fly annotations based on analyzed code properties. From those observations, the stubs can be modified like as follow:

- The first case is when the function may generate buffer overflows, and this function is using a size parameter as an argument. It is here possible to define that the output memory should be valid, and to make sure that there is no buffer overflow. To do so, it is possible to add the following pre-condition:

```
requires <function_name>_security_potential_buffer_overflow: \valid(dest + (0 .. size - 1));
```

So, in the case of the `fgets` function for example, the stub is changed from:

```
/*@
  requires valid_stream: \valid(stream);
  assigns s[0..size] \from indirect:size, indirect:*stream;
  assigns \result \from s, indirect:size, indirect:*stream;
  ensures result_null_or_same: \result == \null || \result == s;
  ensures terminated_string_on_success:
    \result != \null ==> valid_string(s);
*/
extern char *fgets(char * restrict s, int size,
  FILE * restrict stream);
```

To:

```
/*@
  requires fgets_security_potential_buffer_overflow: \valid(s + (0 .. size - 1));
  requires valid_stream: \valid(stream);
  assigns s[0..size] \from indirect:size, indirect:*stream;
  assigns \result \from s, indirect:size, indirect:*stream;
  ensures result_null_or_same: \result == \null || \result == s;
  ensures terminated_string_on_success:
    \result != \null ==> valid_string(s);
*/
extern char *fgets(char * restrict s, int size,
  FILE * restrict stream);
```

This case can be adapted to the following functions: `strncpy`, `memcpy`, `bcopy`, `strncat`.

- In the case the library function may generates buffer overflow, but there is no size parameter in its argument, there is no way to simply create a stub verifying that the size of the destination is superior or equal to the length of the input (especially when the input comes from user supplied string). As such, instead of making formal assertions here, it is possible to use Frama-C to detect the use of such a function (instead of using an additional tool). To do so, one could use a requirement pre-condition that is always false. But this comes at the cost of the function to not be analyzed by the core of Frama-c. In fact, the only proper way to do it is to modify Frama-c shipping or to rewrite every dangerous C functions to include a wrong assert that will emit an alarm during the analysis. The first approach does seem better, as it avoids to maintain a separate codebase from Frama-c and standard Libc. One can find the corresponding modifications in the annex of this document. Those modifications for example permit to emit a warning when the functions `gets` and `toto` are called.

- In the case the function may create potential integer overflows; it seems there is no way to ensure such a thing using E-ACSL. The idea is to use the same modifications than before to emit a warning when a dangerous function is called.

- In the case the function is sensible to format string vulnerabilities, there is also no possibility using E-ACSL to detect them accurately. The same approach than before can be applied, in order to detect them.

4.2.1.2 Generic methodology to make Frama-C analyses

The following paragraph presents a generic approach on how to use Frama-C to analyze a source code in the automated review part of the methodology. This approach is to be followed in the case one reviewer has to use Frama-C on a code. To produce this methodology, the following two principles were followed: the automated review part should be able to detect most of the intrinsic C vulnerabilities, and this review should not take too long to be done. Especially, it was sought to reduce a lot the verbosity of Frama-C. This methodology assumes also that the reviewer knows if the code can be analyzed with Frama-C. Indeed, C Windows code for example has almost no chance to be analyzed with Frama-C, as windows libc is not supported. In the case the code is not complete or use proprietary libraries not shipped with it, due to the soundness of Frama-C, there is a huge probability that the analysis won't be relevant.

This methodology is composed of 6 steps:

- 1) Compiling project with GCC
- 2) Preprocessing files with Frama-C
- 3) Frama-C value analysis
- 4) Results triaging
- 5) Results analysis
- 6) Results refining

4.2.1.2.1 Compiling project with GCC

This step is optional but highly recommended. Indeed, the whole compilation chain might not be provided with the code, and as such simply trying to compile the project will make the reviewer loose too much time.

If possible, compiling the project with GCC, is a complementary approach to the use of Frama-C. Indeed, GCC is by itself capable of detecting several vulnerabilities, like uninitialized variables or types confusions. It is especially useful because it can detect some vulnerabilities not detected by Frama-C, and conversely, like format string errors and several buffer overflows that may occur while calling standard libc functions. It is also a first step to make sure that Frama-C will be able to preprocess the source files.

This compilation needs to be made with several flags to raise more warnings that are interesting in our case. This flag needs to be added to the compilation chain of the source code.

Those flags are the following:

- Wall: Enables almost all compiler's warning messages.
- Wextra: Enables some extra warning messages that are not enabled by -Wall.
- Wnull-dereference: This warning is not set on by the two previous options, and can permit to point out some null dereference vulnerabilities.
- Wformat=2: This option enables more precision while emitting warnings on format strings.
- Wpedantic: Enables some checks like arithmetic overflows.

- Wconversion: Enables some additional checks on type conversions, as it is not enabled by `-Wall` and `-Wextra`.

Once the compilation is ended, every warning should be investigated to check if there is any potential security vulnerability declared with it.

From a practical point of view, compilation chains are often provided with a `Makefile`. A Typical workflow is to open that `Makefile`, and search for the `CPPFLAGS` or `CFLAGS` variable. This variable is modified by adding `"-Wall -Wextra"`. Then the compilation can be launched by a `make all | tee output.txt` command. By adding `tee` command, the output is redirected also into a file, facilitating the process of the warnings emitted.

4.2.1.2.2 Preprocessing files with Frama-C

The next step of the analysis is to preprocess files with Frama-C, that is simply parse them and make sure Frama-C is capable of handling every construct. To do so, the following command should be entered:

```
frama-c -c11 -machdep <the targeted platform for our code> `find . -name *.c`
```

The `-c11` option is added to the command line to allow the use of some C11 constructs that are otherwise prohibited. The idea is to tend to a more global analysis and avoid possible errors that can be simply omitted. Here are two examples of prohibited syntax that are authorized when using the `-c11` option:

```
static const int NUM = 6;

void function(void) {
    char test[NUM];
    char test1[6] = {0};
}
```

In this example, the first array declaration is not correct because of the constant integer passed as initialization size. In the second one, the syntax is prohibited because of the initializer `"{0}"`.

The option `machdep` is used to avoid errors based on the size and alignment of elementary data types.

4.2.1.2.3 Frama-C in-depth analysis

To run the real analysis of Frama-C, the following command line should be used:

```
frama-c -c11 -machdep <arch> -val -no-results -remove-redundant-alarms -
value-log w:<output_file> -val-reduce-on-logic-alarms `find . -name *.c` -
save savefile
```

Once this command is finished, the output file should be gathered for the following steps of the use of Frama-C. Indeed, this command basically asks Frama-C to simply output warnings in the `output_file`, and to reduce redundant alarms the most. The `-no-results` option is used as it is not relevant in our case to keep a trace of the values across the program symbolic execution.

4.2.1.2.4 Triaging results

The next step is to triage a bit the results of the analysis, and especially to remove useless alarms, from a security view point, that may be present within the output file. To do so, the following script might be used on the output file:

```
cat $1| grep -v 'RHS' > temp1
cat temp1| grep -v 'LHS' > temp2
```

```

cat temp2| grep -v 'unknown' > temp3
cat temp3| grep -v 'invalid' > temp4
cat temp4| grep -v '\[value\] warning' > temp5
cat temp5| grep -v 'pointer comparison' > temp6

```

This script does the following:

- removes possibly warning left in the output_file, that are not alarms. Indeed, it appears that only alarms are relevant to security analysis.
- Removes some messages stating on a non-compliance from MISRA development guide. These messages are for example indicating that it is not possible to use non binary fields on binary operators (the remove of lines using RHS and LHS keywords).
- Removes messages stating Frama-C was not able to correctly check some E-ACSL properties. This is the removal based on unknown and invalid keywords.
- Removes messages stating on an error about a pointer comparison. In fact, the comparison might be false from a logical point of view, but it should not create any bug. Those messages could however be used in the manual analysis.

4.2.1.2.5 Analyzing results

The first step for analyzing results is to access the “red alarms” of Frama-c, e.g. those considered as critical or that are analysis errors. To do so, graphical interface should be used with the results of the analysis. To do so, the following command should be used:

```
frama-c-gui -load savefile
```

where savefile is the filename used as a parameter for the save argument of the analysis.

Once in the GUI, the “red alarms” tab should then be accessed, and the alarms emitted here analyzed. Here, a “mem_access” alarm indicates a wrong memory access or allocation, inducing potentially either an overflow or a potential leak of information. An “Initialization” alarm indicates a potential use of an uninitialized variable.

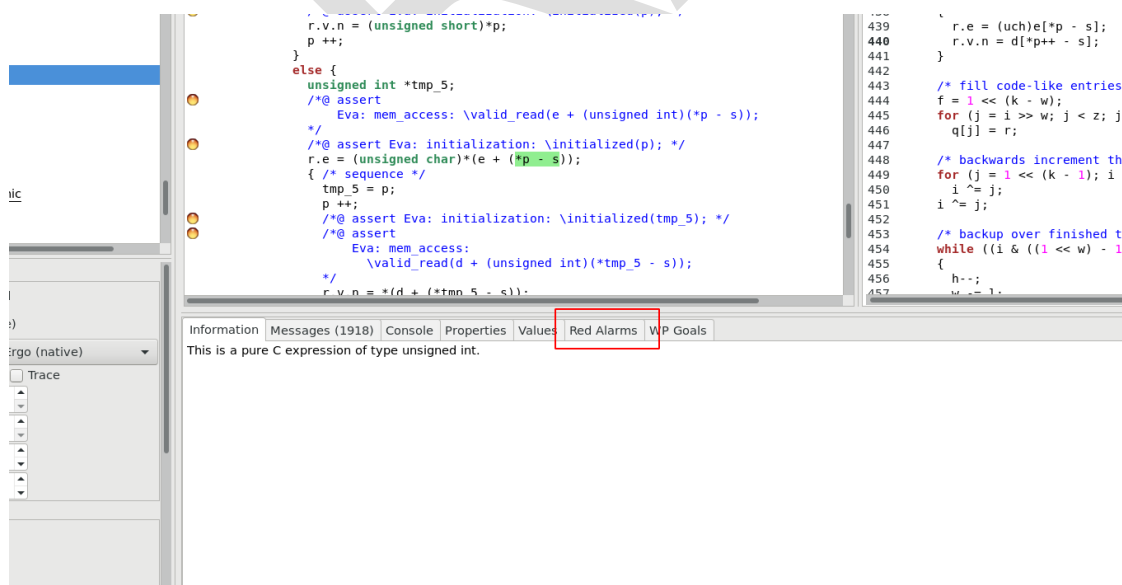


Figure 35: accessing red alarms tab within the GUI of Frama-c

Then, every line of the output file (the one containing all the warnings) should be analyzed. Here is how it can be done:

- If the line contains the following: “out of bounds write”, followed by a potential “\valid(<expression>”. This message means there is potentially a buffer overflow in the application. To determine if the alert is relevant or not, the reviewer needs to check for the size of the buffer in which some data is written, compared to the size of the data written.
- If the line indicates the following: “out of bounds read”. This message indicates a potential information leak. The reviewer needs to make sure that the memory accessed is in fact allocated.
- If the message is: “accessing out of bound index”. This message may indicate a potential of-by-one bug or buffer overflow. Once again, the idea for the reviewer is to check that indeed the buffer access is realized within its bounds or not.
- If the line contains “signed/unsigned overflow”. This message indicates a potential integer overflow problem. The reviewer has to make sure that such an overflow might not occur.
- Another message of interest is “accessing left-value that contains escaping address”. Indeed, this message might indicate a potential use-after-free, or double free vulnerability, because Frama-C cannot make sure the address accessed is valid. To check, a data flow analysis on the variable creating this message should be realized to verify if it is always allocated and correctly accessed.
- The sixth message that may be interesting is “accessing uninitialized left-value”. It may indeed indicate an uninitialized variable use. The reviewer should conduct a data flow analysis to check if the variable is in fact initialized or not.
- Some messages may indicate “underflow”, that means a possible logic error may occur. The reviewer should check if the access variables were not tampered in an incorrect manner.

4.2.1.2.6 Refining results

This step should in fact be conducted at the same time than the previous step. The idea is to be able to realize a quick triaging of results, or to rerun an analysis that may provide better results in the case the number of alerts emitted by Frama-C is too high for the analysis time to fit in the time given by the reviewer. Unfortunately, this step is too dependent on the results of the analysis. Some examples will be given instead.

- In the case the reviewer notes that many alarms are emitted by Frama-C due to a huge context use by the program, he may create a fixed environmental context for the program. To do so, he may create a new `analysis_main` function into the program that initiates the whole context of the analysis, and declare this function as the new entry point for the analysis.

Here is an example taken from the documentation of Frama-C. First, let's imagine the following program (in this case, Frama-C will complain on not being able to provide correct interval to `argv`) :

```
int main(int argc, char **argv)
{
    if (argc != 2) usage();
    //following of the program
    ...
}

void usage(void)
{
    printf("this application expects an argument between '0' and '9' \n");
    exit(1);
}
```

Based on the information provided by this source code, one can understand that the program waits for one argument that is a numeric value. As such, the `analysis_main` function can be written like this:

```
int analysis_main(void)
{
    char *argv[3];
    argv[0]="Analyzed application";
    argv[1]="1";
    argv[2]=NULL;
    return main(2, argv);
}
```

The same can be done for global variables or environment ones.

- In the case the reviewer notes that a lot of errors are due to the use of a macro, the macro can be rewritten to be error safe, or some E-ACSL annotations can be added to add to the verification.

- Another way to reduce the number of alarms is to annotate all loops in the code that can be easily determined. The idea is to check every “for” loops in the code, and check if the boundary can be determined. If so, an assertion inducing an unrolling of the loop will increase the precision of the results, which should reduce the number of false positive alarms. This assertion is formed like that:

```
/*@ loop pragma UNROLL <number of loops>; */
```

Here is an example showing how to use such an assertion. Let’s consider the following code:

```
#include <stdlib.h>
int main() {
    int * tab=(int *) malloc(sizeof(int)*10);
    int i;
    tab[0]=1;
    for ( i=0;i<11; i++)
        tab[i]=1;
    return 0;
}
```

One can see there is a “for” loop within the code, where the boundary is “11”. The “for” loop will be executed as such ten times. In order to have better results of Frama-c, the code can be modified like this:

```
#include <stdlib.h>
int main() {
    int * tab=(int *) malloc(sizeof(int)*10);
    int i;
    tab[0]=1;
    /*@ loop pragma UNROLL 10; */
    for ( i=0;i<11; i++)
        tab[i]=1;
    return 0;
}
```

- Also, if there is too much of the use of some particular constructs in the code, like the use of dangling pointers between functions, it might be interesting to simply drop every alarms emitted by Frama-C associated with that, for example by running `grep -v` on the output file.
- Finally, if there are too many alarms still left, it might be interesting to consider that the code is not really analyzable by Frama-c. A less precise and sound tool can so be used instead.

4.2.2 Using Frama-C on the manual review part of the analysis phase

Actually, Frama-C could also be used during the manual review of the analysis phase. Indeed, its formal analysis led to the creation of two plugins that may be interesting in our case, because one of the major parts of the manual review is to be able for the reviewer to navigate in the source code given the data flow, and Frama-C is able to construct this data flow.

For example, the reviewer may stumble upon a call to a “strcpy” that may overflow the resulting buffer, without knowing how the resulting buffer was allocated in the code. To do so, he has to go to the top of the control flow of the program, to search for the allocation possibilities.

In another part, the reviewer may want to know how the variable that controls the authentication in a given program is handled. To do so, the idea is to know what are the statements affected by the modification of this variable in the program.

The first plugin that is of interest is the impact plugin:

```

int b3(int *i)
{
    int huhu;
    huhu = *i;
    huhu ++;
    printf va 2("coucou b %i\n", huhu);
    return huhu;
}
  
```

```

test_slicing3.c
5 void a3(){
6   printf("coucou a\n");
7 }
8
9 void a2(){
10  a3();
11 }
12
13 void a1(){
14  a2();
15 }
16
17 void a(){
18  a1();
19 }
  
```

Figure 36: using Impact plugin in the graphical interface of Frama-C (green statements are the one highlighted by Impact plugin)

Basically, this plugin highlights all statements that are affected as a side effect of a given statement of a C program. That means all statements where data flow is modified, and all statements in which the statement is used.

The second plugin of interest is the Scope one, using its zones function:

```

int b3(int *i)
{
    int huhu;
    huhu = *i;
    huhu ++;
    printf va 2("coucou b %i\n", huhu);
    return huhu;
}
  
```

```

test_slicing3.c
17 void a(){
18  a1();
19 }
20
21
22 int b3(int* i){
23  int huhu;
24  huhu = *i;
25  huhu +=1;
26  printf("coucou b %i\n", huhu);
27  return huhu;
28 }
29
30 int b2(){
31  int az = 2;
  
```

Figure 37: using Zones to highlight statements (in pink) that define the value of huhu variable in the printf function

This plugin highlights statements in a function that impact the value D of a variable at a certain point L . It can be interesting in the case big functions are present in the code, to be able to quickly identify the statements responsible for the value a variable at a point.

The two following plugins can be used like the following: the reviewer detects a point in the code from where a data flow analysis could be interesting. He then uses the two plugins to be able to conduct this data flow analysis, and to highlights the statements responsible for the values at this point of the program. It is a way for him to avoid to analyze useless code.

However, whilst those plugins are of interest, a certain number of limitations are present: for Impact plugin, there is no clear view of the affected functions. The reviewer needs to navigate by hand between each function to discover which one is affected. Some buttons to navigate between the impacted statements, given the dataflow of the program, could be interesting there. In batch processing, statements impacted are printed without any context apart from the line number. What could be interesting there is that impact prints the affected statements given the data flow of the program. Also, `Scope` plugin do not expand past the caller.

So, in general, Frama-C can be used for the manual review part of the analysis phase in the case the code is not too big, and can be understood easily for navigation.

Chapter 5 Applying Frama-C's use on an example

This paragraph aims at explaining how to use Frama-C concretely on a real-life example, given the methodology presented at the previous chapter. The idea is so to understand how to possibly adapt quickly Frama-C's use to the code analyzed in order to get better results.

5.1 Choosing a sample

For the example to be chosen, several criteria have been selected:

- The code should be of a well-known tool
- The code should be known to be vulnerable
- Frama-C should be usable on the code without the need to rewrite huge portions of it

After several searches, it appears that gzip 1.2.4 might be a good candidate. Indeed, several CVE have been posted on it, and the code appears in the github repository "open-source-case-studies" of Frama-C, used for benchmarking. Moreover, gzip is currently a tool used on every linux distribution. The sample used for the following can be found here: <https://ftp.gnu.org/gnu/gzip/gzip-1.2.4.tar>. From the `Makefile`, it appears the code of the Linux version of Gzip is composed of 7324 LoC.

5.2 Quick discovery phase

This example has no thorough documentation provided. As a starting point, one can however take the currently available information on gzip given in:

- the man page: <http://manpagesfr.free.fr/man/man1/gzip.1.html>,
- the readme of the project (as well as the NEWS file, and the Changelog),
- the TODO file, as it may contain information on vulnerabilities to be corrected in the future.

Gzip is not a critical application, as all it does is basically compressing files and decompressing them. It is also almost always used with the privileges of the current user. There are two risks associated with the use of this tool:

- A bug can be triggered in Gzip to alter its execution flow. This can be dangerous in the case Gzip is for example run on a file storage server. It will indeed provide a remote access to an attacker. A deny of service is however not interesting in any case, because gzip won't create a deny of service of the whole server.
- In case gzip is run with other privileges (because permissions are handled by the underlying OS), a leak or the rewrite of interesting files could occur. However, this has very low chance to happen as gzip is not run with other privileges in most cases.

From this quick discovery, one can already states on the relevance of the results from using Frama-C: the execution flow alteration might be discovered by Frama-C itself, as it will certainly come from a memory problem. The malicious files manipulation risk however cannot be detected with Frama-C, because this is a race conditions vulnerability or environment interaction one.

5.3 Review phase

5.3.1 Automated review

Once again, we will step through every steps required for the automated review here. The version of Frama-C in use here is Frama-C Chlorine (version 17). No stubs modification has been done for this analysis.

1) Compiling the project

Gzip is shipped with a `Makefile.in` and a `configure` script. To create a Makefile, the command `./configure` is launched into the directory of Gzip. Then, to compile it with the additional options described in the previous chapter, the Makefile is modified as such:

```
- CFLAGS = -O
+ CFLAGS = -O -Wall -Wextra -Wpedantic -Wformat=2 -Wnull-dereference -
Wconversion
```

The compilation is then launched. The output is around 200 warnings emitted, and they all concern possible faults from converting from one type to another. However, most of them appear to be irrelevant, as the numbers manipulated are too far from the bounds.

2) Preprocessing files

Files are then preprocessed with Frama-C, given the following command:

```
frama-c -c11 -machdep x86_64 gzip.c zip.c deflate.c trees.c bits.c unzip.c
inflate.c util.c crypt.c lzw.c unlz.c unpack.c unlzh.c getopt.c
```

Here, the list of files provided is taken from the Makefile. Indeed, a lots of C files present in the repository are here for cross-platform support and are not needed (and won't certainly be correctly analysed by Frama-C). So, only the relevant ones are included for the preprocessing.

The output of this command is the following :

```
[kernel] Parsing gzip.c (with preprocessing)
[kernel:typing:incompatible-types-call] gzip.c:448: Warning:
  implicit conversion between incompatible function types:
  void (*)(void)
  and
  void (*)(int )
[kernel:typing:incompatible-types-call] gzip.c:452: Warning:
  implicit conversion between incompatible function types:
  void (*)(void)
  and
  void (*)(int )
[kernel:typing:incompatible-types-call] gzip.c:457: Warning:
  implicit conversion between incompatible function types:
  void (*)(void)
  and
  void (*)(int )
[kernel:typing:implicit-function-declaration] gzip.c:611: Warning:
```

```

    Calling undeclared function isatty. Old style K&R code?
[kernel:typing:implicit-function-declaration] gzip.c:778: Warning:
    Calling undeclared function close. Old style K&R code?
[kernel:typing:implicit-function-declaration] gzip.c:831: Warning:
    Calling undeclared function unlink. Old style K&R code?
[kernel:typing:implicit-function-declaration] gzip.c:1369: Warning:
    Calling undeclared function lseek. Old style K&R code?
[kernel:typing:implicit-function-declaration] gzip.c:1373: Warning:
    Calling undeclared function read. Old style K&R code?
[kernel:typing:implicit-function-declaration] gzip.c:1632: Warning:
    Calling undeclared function chown. Old style K&R code?
[kernel] Parsing zip.c (with preprocessing)
[kernel:typing:implicit-function-declaration] zip.c:111: Warning:
    Calling undeclared function read. Old style K&R code?
[kernel] Parsing deflate.c (with preprocessing)
[kernel] Parsing trees.c (with preprocessing)
[kernel] Parsing bits.c (with preprocessing)
[kernel] Parsing unzip.c (with preprocessing)
[kernel] Parsing inflate.c (with preprocessing)
[kernel] Parsing util.c (with preprocessing)
[kernel:typing:implicit-function-declaration] util.c:46: Warning:
    Calling undeclared function read. Old style K&R code?
[kernel:typing:implicit-function-declaration] util.c:156: Warning:
    Calling undeclared function write. Old style K&R code?
[kernel] Parsing crypt.c (with preprocessing)
[kernel] Parsing lzw.c (with preprocessing)
[kernel] Parsing unlzw.c (with preprocessing)
[kernel:typing:implicit-function-declaration] unlzw.c:261: Warning:
    Calling undeclared function read. Old style K&R code?
[kernel] Parsing unpack.c (with preprocessing)
[kernel] Parsing unlzh.c (with preprocessing)
[kernel] Parsing getopt.c (with preprocessing)
[kernel] User Error: Incompatible declaration for read:
    different integer types unsigned long and unsigned int
    First declaration was at gzip.c:1373
    Current declaration is at zip.c:111
[kernel] Frama-C aborted: invalid user input.

```

A study of this output shows that Frama-C is not able to find the definition of several functions, whilst GCC does. That means there is certainly some compilation options at the source of the problems of Frama-C. After a bit of research, it appears the Makefile contains the following flags: `-DSTDC_HEADERS=1 -DHAVE_UNISTD_H=1 -DDIRENT=1 -DNO_UTIME=1`.

The command invoking Frama-C can be modified to include those:

```
frama-c -c11 -machdep x86_64 -cpp-extra-args='-DSTDC_HEADERS=1 -
DHAVE_UNISTD_H=1 -DDIRENT=1 -DNO_UTIME=1' gzip.c zip.c deflate.c trees.c
bits.c unzip.c inflate.c util.c crypt.c lzw.c unlzw.c unpack.c unlzh.c
getopt.c
```

The only error appearing in the output is then the following:

```
[kernel] User Error: Incompatible declaration for strcmp: different integer
types unsigned long and int
First declaration was at FRAMAC_SHARE/libc/string.h:130
Current declaration is at getopt.c:175
```

To correct this error, the following modification is done in getopt.c:

```
- extern int  strcmp(const char *s1, const char *s2, int n);
+ extern int  strcmp(const char *s1, const char *s2, size_t n);
```

There are warnings left, but the processing does not output any more error.

3) Analysis and triaging

The analysis is run with the following command:

```
frama-c -c11 -machdep x86_64 -cpp-extra-args='-DSTDC_HEADERS=1 -
DHAVE_UNISTD_H=1 -DDIRENT=1 -DNO_UTIME=1' -val -no-results -remove-redundant-
alarms -value-log w:<output_file> -val-reduce-on-logic-alarms gzip.c zip.c
deflate.c trees.c bits.c unzip.c inflate.c util.c crypt.c lzw.c unlzw.c
unpack.c unlzh.c getopt.c -save <savefile>
```

The first analysis terminates with an error stating a degeneration point reached by the EVA plugin. The option `-val-ignore-recursive-calls` is used to be able to conduct the analysis. It takes approximately 1h and the output is 2807 lines long in the log file. A count of the keyword `alarm` reveals 1669 unique alarms. Once passed to the triaging script, the final output is composed of 1382 alarms.

4) Results review

Some attempts were made to reduce even more the number of alarms emitted by EVA plugin. A quick way is to create a context for the main function. While doing this, the number of alarms is reduced by several dozens. It was attempted also to make few modifications on macros to add type casts when it was not present. The few modifications realized did not modify the total number of alarms. In the end, there are still approximately one thousand alarms to be checked by the reviewer.

This is a too big number to be considered like a quick-win phase of the review. In fact, checking every alarm might be here not more efficient that making the whole review by hand. As such, the code analysed here might not be appropriate for an analysis with Frama-c. An interesting feature would be to provide a score associated with an alarm. The idea would be to indicate the “correctness” of one alarm (for example, if an alarm is emitted based on an approximation realized by Frama-C, then the score is low).

However, instead of checking every alarm, it was searched if Frama-C was able to detect the CVE that are depending on bugs in the code:

- CVE-2010-0001: Integer underflow in the unlzw function in unlzw.c in gzip before 1.4 on 64-bit platforms, as used in ncompress and probably others, allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a crafted archive that uses LZW compression, leading to an array index error. If one looks at the patch:

```
diff --git a/unlzw.c b/unlzw.c
```



```

index fb9ff76..8f8cbee 100644
--- a/unlzw.c
+++ b/unlzw.c
@@ -240,7 +240,8 @@ int unlzw(in, out)
    int o;

    resetbuf:
-   e = insize-(o = (posbits>>3));
+   o = posbits >> 3;
+   e = o <= insize ? insize - o : 0;

    for (i = 0 ; i < e ; ++i) {
        inbuf[i] = inbuf[i+o];

```

It seems that `e`, which should be a positive integer, might indeed get a negative value if `o` is bigger than `insize`. Then, the variable `e` is used to indicate the offset where to read in the input file. As such, trying to read a negative offset in the input file will trigger the crash. Here, Frama-C gives several warnings of integer overflow. It could be used to potentially detect this vulnerability.

- CVE-2009-2624: it appears `huft_build` function in `inflate.c` in `gzip` before 1.3.13 creates a hufts (aka huffman) table that is too small. When looking at the patch, here is the correction applied:

```

diff --git a/inflate.c b/inflate.c
index 7dd630a..2f8670d 100644
--- a/inflate.c
+++ b/inflate.c
@@ -335,13 +335,15 @@ int *m;          /* maximum lookup bits, returns
actual */
    } while (--i);
    if (c[0] == n)          /* null input--all zero length codes */
    {
-   q = (struct huft *) malloc (2 * sizeof *q);
+   q = (struct huft *) malloc (3 * sizeof *q);
    if (!q)
        return 3;
-   hufts += 2;
+   hufts += 3;
    q[0].v.t = (struct huft *) NULL;
    q[1].e = 99;          /* invalid code marker */
    q[1].b = 1;
+   q[2].e = 99;          /* invalid code marker */
+   q[2].b = 1;
    *t = q + 1;
    *m = 1;

```

```
return 0;
```

One can understand that a heap overflow may be created within the application, in the case the archive inflated is specially crafted for. Unfortunately, this vulnerability cannot be trivially determined. Frama-C warns about a potential buffer overflow once the `q` is written into:

```
inflate.c:446:[value:alarm] warning: out of bounds write. assert \valid(q + j_0);
```

As such, there is a possibility that Frama-C detects correctly the vulnerability here.

- CVE-2001-1228: Buffer overflows in gzip 1.3x, 1.2.4, and other versions might allow attackers to execute code via a long file name, possibly remotely if gzip is run on an FTP server. The patch is the following:

```
--- gzip.c Thu Aug 19 09:39:43 1993
+++ gzip-fix.c Sun Dec 30 13:57:44 2001
@@ -1006,7 +1006,7 @@
char *dot; /* pointer to ifname extension, or NULL */
#endif

- strcpy(ifname, iname);
+ strncpy(ifname, iname, sizeof(ifname) - 1);

/* If input file exists, return OK. */
if (do_stat(ifname, sbuf) == 0) return OK;
@@ -1683,7 +1683,7 @@
}
len = strlen(dir);
if (len + NLENGTH(dp) + 1 < MAX_PATH_LEN - 1) {
- strcpy(nbuf, dir);
+ strncpy(nbuf, dir, sizeof(nbuf) - 1);
if (len != 0 /* dir = "" means current dir on Amiga */
#ifdef PATH_SEP2
&& dir[len-1] != PATH_SEP2
```

Here, a buffer overflow can be triggered in the BSS by the use of a very long input file (in fact, today, given limitations on command line, this bug cannot be triggered). The output of Frama-C reveals however that this vulnerability is not detected while using the common shipping. In the case the stub of the function is modified however, the vulnerability could have been identified.

Finally, Frama-C appears interesting to detect potential vulnerabilities in the source code. Unfortunately, its actual output is too much to be relevant for security reviewers.

5.3.2 Manual review phase

As it is not easy to explain how Frama-C Impact and Scopes plugin can be used for this part, it was intended to show an example on how to use those plugins to get back to the CVE-2005-0988. This CVE concerns the possibility to modify permissions of arbitrary files when

decompressing, via a hard-link attack, within `setuid` directories. This can be used for example to obtain a readable file that was not.

However, once this CVE has been analyzed, it appears that it cannot be really detected given scopes plugin (it was hoped that a handle to the same file is used, but here `istat` structure is used in the meantime).

Unfortunately, no other CVE published on Gzip can be confirmed by the two plugins highlighted previously.

DRAFT

Chapter 6 Summary and Conclusion

This document has presented a generic security source code auditing methodology, as well as C memory management and several common vulnerabilities depending on it. Using Frama-C for such an audit has been discussed then, from a generic point of view as well as on a real open-source example.

For vulnerabilities detection, Frama-C seems interesting in the case of narrowed and well-mastered code, which can be found in embedded systems. However, in the case of codes having a huge number of interactions with their environment, using Frama-C can be more difficult.

Several improvements can be made to Frama-C in order to be able to analyse more codes, from stubs development (and maybe E-ACSL) to dedicated plugins for security reviews. VESSEDIA project finally appears to be good candidate to provide such improvements.

DRAFT

Chapter 7 List of Abbreviations

Abbreviation	Translation
API	Application programming interface
CMM	Capability Mature Model
CPU	Control processing unit
CVE	Common vulnerability exposure
DoS	Deny of Service
I/O	Input/Output
IoT	Internet of Things
LoC	Lines of Code
OS	Operating System
SDLC	Software development life-cycle
WP	Work Package

Chapter 8 Bibliography

- [1] "Managing the Software Process", Watts Humphrey
- [2] "KeePass Password Safe, code review results report",
https://joinup.ec.europa.eu/sites/default/files/inline-files/DLV%20WP6%2001-%20KeePass%20Code%20Review%20Results%20Report_published.pdf
- [3] "ITIL Version 3 Service Lifecycle for Application Support", <https://www.fichier-pdf.fr/2011/06/16/itil-v3-application-support/>
- [4] "ISO/IEC 27034", <https://www.iso.org/en/standard/44378.html>
- [5] "NIST SP 800-37/64", <https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final>
- [6] "SOG-IS Agreed Cryptographic Mechanisms",
https://www.sogis.org/uk/supporting_doc_en.html
- [7] Owasp code review project,
https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents
- [8] SOG-IS agreed cryptographic mechanisms,
https://www.sogis.org/uk/supporting_doc_en.html
- [9] MISRA-C coding baselines, <https://www.misra.org.uk/Publications/tabid/57/Default.aspx>

DRAFT


```

Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = string

module EqualityStorage: Parameter_sig.Bool
module SymbolicLocsStorage: Parameter_sig.Bool
module GaugesStorage: Parameter_sig.Bool
module ApronStorage: Parameter_sig.Bool
module BitwiseOffsmStorage: Parameter_sig.Bool

module AutomaticContextMaxDepth: Parameter_sig.Int
module AutomaticContextMaxWidth: Parameter_sig.Int

module AllRoundingModesConstants: Parameter_sig.Bool

module NoResultsFunctions: Parameter_sig.Fundec_set
module ResultsAll: Parameter_sig.Bool

module JoinResults: Parameter_sig.Bool

module WarnSignedConvertedDowncast: Parameter_sig.Bool
module WarnPointerSubstraction: Parameter_sig.Bool
module WarnCopyIndeterminate: Parameter_sig.Kernel_function_set

module IgnoreRecursiveCalls: Parameter_sig.Bool

module SemanticUnrollingLevel: Parameter_sig.Int
module SlevelFunction:
  Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = int

module SlevelMergeAfterLoop: Parameter_sig.Kernel_function_set

module MinLoopUnroll : Parameter_sig.Int

module DescendingIteration: Parameter_sig.String
module HierarchicalConvergence: Parameter_sig.Bool
module WideningDelay: Parameter_sig.Int
module WideningPeriod: Parameter_sig.Int
module ArrayPrecisionLevel: Parameter_sig.Int

module AllocatedContextValid: Parameter_sig.Bool
module InitializationPaddingGlobals: Parameter_sig.String

module SaveFunctionState:
  Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = string
module LoadFunctionState:
  Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = string
val get_SaveFunctionState : unit -> Cil_types.kernel_function * string
val get_LoadFunctionState : unit -> Cil_types.kernel_function * string

module Numerors_Real_Size : Parameter_sig.Int
module Numerors_Mode : Parameter_sig.String

module UndefinedPointerComparisonPropagateAll: Parameter_sig.Bool
module WarnPointerComparison: Parameter_sig.String

module ReduceOnLogicAlarms: Parameter_sig.Bool
module InitializedLocals: Parameter_sig.Bool

```



```

module UsePrototype: Parameter_sig.Kernel_function_set

module SkipLibcSpecs: Parameter_sig.Bool

module RmAssert: Parameter_sig.Bool

module LinearLevel: Parameter_sig.Int
module BuiltinsOverrides:
  Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = string
module BuiltinsAuto: Parameter_sig.Bool
module BuiltinsList: Parameter_sig.Bool
module SplitReturnFunction:
  Parameter_sig.Map with type key = Cil_types.kernel_function
                    and type value = Split_strategy.t
module SplitGlobalStrategy: State_builder.Ref with type data = Split_strategy.t

module ValShowProgress: Parameter_sig.Bool
module ValShowInitialState: Parameter_sig.Bool
module ValShowPerf: Parameter_sig.Bool
module ValPerfFlamegraphs: Parameter_sig.String
module ShowSlevel: Parameter_sig.Int
module PrintCallstacks: Parameter_sig.Bool
module AlarmsWarnings: Parameter_sig.Bool
module ReportRedStatuses: Parameter_sig.String
module NumerorsLogFile: Parameter_sig.String
module WarnBuiltinOverride: Parameter_sig.Bool

module MemExecAll: Parameter_sig.Bool

module InterpreterMode: Parameter_sig.Bool
module StopAtNthAlarm: Parameter_sig.Int

(** Dynamic allocation *)

module MallocFunctions: Parameter_sig.String_set
module AllocReturnsNull: Parameter_sig.Bool
module MallocLevel: Parameter_sig.Int

val parameters_correctness: Typed_parameter.t list
val parameters_tuning: Typed_parameter.t list
val parameters_abstractions: Typed_parameter.t list

(** Debug categories responsible for printing initial and final states of Value.
    Enabled by default, but can be disabled via the command-line:
    -value-msg-key="-initial_state,-final_state" *)
val dkey_initial_state : category
val dkey_final_states : category

(** Warning category used when emitting an alarm in "warning" mode. *)
val wkey_alarm: warn_category

(** Warning category used for the warning "locals escaping scope". *)
val wkey_locals_escaping: warn_category

(** Warning category used to print garbled mix *)
val wkey_garbled_mix: warn_category

(** Warning category used for "cannot use builtin due to missing spec" *)

```

```

val wkey_builtins_missing_spec: warn_category

(** Warning category used for "definition overridden by builtin" *)
val wkey_builtins_override: warn_category

(** Warning category used for calls to libc functions whose specification
    is currently unsupported. *)
val wkey_libc_unsupported_spec : warn_category

(** Warning category used for "loop not completely unrolled" *)
val wkey_loop_unrolling : warn_category

(* ABP *)
(** Warning category used for "dangerous functions traversal" *)
val wkey_dangerous_functions : warn_category

(** Debug category used to print information about invalid pointer comparisons*)
val dkey_pointer_comparison: category

(** Debug category used to print the cvalue domain on Frama_C_[dump|show]_each
    functions. *)
val dkey_cvalue_domain: category

(* Print non-bottom product of states with no concretization, revealed by
    an evaluation leading to bottom without alarms. *)
val dkey_incompatible_states: category

(** Debug category used to print information about the iteration *)
val dkey_iterator : category

(** Debug category used when using Eva callbacks when recording the results of
    a function analysis. *)
val dkey_callbacks : category

(** Debug category used to print the usage of widenings. *)
val dkey_widening : category

(*
Local Variables:
compile-command: "make -C ../../.."
End:
*)

```

- File value_parameters.ml

```

(*****)
(* *)
(* This file is part of Frama-C. *)
(* *)
(* Copyright (C) 2007-2018 *)
(* CEA (Commissariat à l'énergie atomique et aux énergies *)
(* alternatives) *)
(* *)
(* you can redistribute it and/or modify it under the terms of the GNU *)
(* Lesser General Public License as published by the Free Software *)
(* Foundation, version 2.1. *)
(* *)
(* It is distributed in the hope that it will be useful, *)
(* but WITHOUT ANY WARRANTY; without even the implied warranty of *)

```

```

(* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the      *)
(* GNU Lesser General Public License for more details.                  *)
(*                                                                      *)
(* See the GNU Lesser General Public License version 2.1                *)
(* for more details (enclosed in the file licenses/LGPLv2.1).          *)
(*                                                                      *)
(* ***** *)

(* Dependencies to kernel options *)
let kernel_parameters_correctness = [
  Kernel.MainFunction.parameter;
  Kernel.LibEntry.parameter;
  Kernel.AbsoluteValidRange.parameter;
  Kernel.SafeArrays.parameter;
  Kernel.UnspecifiedAccess.parameter;
  Kernel.SignedOverflow.parameter;
  Kernel.UnsignedOverflow.parameter;
  Kernel.LeftShiftNegative.parameter;
  Kernel.RightShiftNegative.parameter;
  Kernel.SignedDowncast.parameter;
  Kernel.UnsignedDowncast.parameter;
]

let parameters_correctness = ref Typed_parameter.Set.empty
let parameters_tuning = ref Typed_parameter.Set.empty
let add_dep p =
  State_dependency_graph.add_codedependencies
    ~onto:Db.Value.self
    [State.get p.Typed_parameter.name]
let add_correctness_dep p =
  if Typed_parameter.Set.mem p !parameters_correctness then
    Kernel.abort "adding correctness parameter %a twice"
      Typed_parameter.pretty p;
  add_dep p;
  parameters_correctness := Typed_parameter.Set.add p !parameters_correctness
let add_precision_dep p =
  if Typed_parameter.Set.mem p !parameters_tuning then
    Kernel.abort "adding tuning parameter %a twice"
      Typed_parameter.pretty p;
  add_dep p;
  parameters_tuning := Typed_parameter.Set.add p !parameters_tuning

let () = List.iter add_correctness_dep kernel_parameters_correctness

include Plugin.Register
(struct
  let name = "Eva"
  let shortname = "eva"
  let help =
    "automatically computes variation domains for the variables of the
program"
  end)

let () = Help.add_aliases [ "-value-h"; "-val-h" ]
let () = add_plugin_output_aliases [ "value" ]

(* Debug categories. *)
let dkey_initial_state = register_category "initial-state"
let dkey_final_states = register_category "final-states"
let dkey_pointer_comparison = register_category "pointer-comparison"
let dkey_cvalue_domain = register_category "d-cvalue"
let dkey_incompatible_states = register_category "incompatible-states"

```

```

let dkey_iterator = register_category "iterator"
let dkey_callbacks = register_category "callbacks"
let dkey_widening = register_category "widening"

let () =
  let activate dkey = add_debug_keys dkey in
  List.iter activate
    [dkey_initial_state; dkey_final_states; dkey_cvalue_domain]

(* Warning categories. *)
let wkey_alarm = register_warn_category "alarm"
let wkey_locals_escaping = register_warn_category "locals-escaping"
let wkey_garbled_mix = register_warn_category "garbled-mix"
let () = set_warn_status wkey_garbled_mix Log.Winactive
let wkey_builtins_missing_spec = register_warn_category "builtins:missing-spec"
let wkey_builtins_override = register_warn_category "builtins:override"
let wkey_libc_unsupported_spec = register_warn_category "libc:unsupported-spec"
let wkey_loop_unrolling = register_warn_category "loop-unrolling"
(* ABP : add 1 line *)
let wkey_dangerous_functions = register_warn_category "dangerous_functions"
let () = set_warn_status wkey_loop_unrolling Log.Wfeedback

module ForceValues =
  WithOutput
  (struct
    let option_name = "--eva"
    let help = "compute values"
    let output_by_default = true
  end)
let () = ForceValues.add_aliases ["-val"]

let domains = add_group "Abstract Domains"
let precision_tuning = add_group "Precision vs. time"
let initial_context = add_group "Initial Context"
let performance = add_group "Results memoization vs. time"
let interpreter = add_group "Deterministic programs"
let alarms = add_group "Propagation and alarms "
let malloc = add_group "Dynamic allocation"

(* ----- *)
(* --- Eva domains --- *)
(* ----- *)

(* Set of parameters defining the abstractions used in an Eva analysis. *)
let parameters_abstractions = ref Typed_parameter.Set.empty

(* This functor must be used to create parameters for new domains of Eva. *)
module Domain_Parameter
  (X:sig include Parameter_sig.Input val default: bool end)
= struct
  Parameter_customize.set_group domains;
  module Parameter = Bool (X);;
  add_precision_dep Parameter.parameter;
  parameters_abstractions :=
    Typed_parameter.Set.add Parameter.parameter !parameters_abstractions;
  include Parameter
end

module CvalueDomain = Domain_Parameter
  (struct
    let option_name = "--eva-cvalue-domain"
    let help = "Use the default domain of eva."
  end)

```

```

    let default = true
  end)

module EqualityDomain = Domain_Parameter
  (struct
    let option_name = "-eva-equality-domain"
    let help = "Use the equality domain of Eva."
    let default = false
  end)

module GaugesDomain = Domain_Parameter
  (struct
    let option_name = "-eva-gauges-domain"
    let help = "Use the gauges domain of Eva."
    let default = false
  end)

module SymbolicLocsDomain = Domain_Parameter
  (struct
    let option_name = "-eva-symbolic-locations-domain"
    let help = "Use a dedicated domain for symbolic equalities."
    let default = false
  end)

module BitwiseOffsmDomain = Domain_Parameter
  (struct
    let option_name = "-eva-bitwise-domain"
    let help = "Use the bitwise abstractions of Eva."
    let default = false
  end)

module NumerorsDomain = Domain_Parameter
  (struct
    let option_name = "-eva-numerors-domain"
    let help = "Experimental. Use the numerors domain of Eva. This domain \
      computes rounding error bounds for the floating point \
      computations"
    let default = false
  end)

let apron_help = "Experimental binding of the numerical domains provided \
  by the APRON library: http://apron.cri.ensmp.fr/library \n"

module ApronOctagon = Domain_Parameter
  (struct
    let option_name = "-eva-apron-oct"
    let help = apron_help ^ "Use the octagon domain of apron."
    let default = false
  end)

module ApronBox = Domain_Parameter
  (struct
    let option_name = "-eva-apron-box"
    let help = apron_help ^ "Use the box domain of apron."
    let default = false
  end)

module PolkaLoose = Domain_Parameter
  (struct
    let option_name = "-eva-polka-loose"
    let help = apron_help ^ "Use the loose polyhedra domain of apron."
    let default = false
  end)

```

```

end)

module PolkaStrict = Domain_Parameter
  (struct
    let option_name = "-eva-polka-strict"
    let help = apron_help ^ "Use the strict polyhedra domain of apron."
    let default = false
  end)

module PolkaEqualities = Domain_Parameter
  (struct
    let option_name = "-eva-polka-equalities"
    let help = apron_help ^ "Use the linear equalities domain of apron."
    let default = false
  end)

module InoutDomain = Domain_Parameter
  (struct
    let option_name = "-eva-inout-domain"
    let help = "Compute inputs and outputs within Eva. Experimental."
    let default = false
  end)

module SignDomain = Domain_Parameter
  (struct
    let option_name = "-eva-sign-domain"
    let help = "Use the sign domain of Eva. For demonstration purposes only."
    let default = false
  end)

module PrinterDomain = Domain_Parameter
  (struct
    let option_name = "-eva-printer-domain"
    let help = "Use the printer domain of eva. Useful for the developpers \
of new abstract domains, as it prints the domain functions \
that are called by Eva during an analysis."
    let default = false
  end)

let () = Parameter_customize.set_group domains
module EqualityCall =
  String
  (struct
    let option_name = "-eva-equality-through-calls"
    let help = "Equalities propagated through function calls (from the caller \
\
to the called function): none, only equalities between formal \
\
parameters and concrete arguments, or all. "
    let default = "formals"
    let arg_name = "none|formals|all"
  end)
let () = add_precision_dep EqualityCall.parameter

let () = Parameter_customize.set_group domains
module EqualityCallFunction =
  Kernel_function_map
  (struct
    include Datatype.String
    type key = Cil_types.kernel_function
    let of_string ~key:_ ~prev:_ = function
      | None | Some ("none" | "formals" | "all") as x -> x
  end)

```

```

    | _ -> raise (Cannot_build "must be 'none', 'formals' or 'all'.")
    let to_string ~key:_ s = s
  end)
  (struct
    let option_name = "-eva-equality-through-calls-function"
    let help = "Equalities propagated through calls to specific functions. \
              Overrides -eva-equality-call."
    let default = Kernel_function.Map.empty
    let arg_name = "f:none|formals|all"
  end)
let () = add_precision_dep EqualityCallFunction.parameter

let () = Parameter_customize.set_group domains
module Numerors_Real_Size =
  Int
  (struct
    let default = 128
    let option_name = "-eva-numerors-real-size"
    let arg_name = "n"
    let help =
      "set <n> as the significand size of the MPFR representation \
      of reals used by the numerors domain (defaults to 128)"
  end)
let () = add_precision_dep Numerors_Real_Size.parameter

let () = Parameter_customize.set_group domains
module Numerors_Mode =
  String
  (struct
    let option_name = "-eva-numerors-interaction"
    let help = "defines how the numerors domain infers the absolute and the \
              relative errors:\n\
              - relative: the relative is deduced from the absolute;\n\
              - absolute: the absolute is deduced from the relative;\n\
              - none: absolute and relative are computed separately;\n\
              - both: reduced product between absolute and relative."
    let default = "both"
    let arg_name = "relative|absolute|none|both"
  end)
let () =
  Numerors_Mode.set_possible_values ["relative"; "absolute"; "none"; "both"]
let () = add_precision_dep Numerors_Mode.parameter

(* ----- *)
(* --- Performance options --- *)
(* ----- *)

let () = Parameter_customize.set_group performance
module NoResultsFunctions =
  Fundec_set
  (struct
    let option_name = "-eva-no-results-function"
    let arg_name = "f"
    let help = "do not record the values obtained for the statements of \
              function f"
  end)
let () = add_dep NoResultsFunctions.parameter
let () = NoResultsFunctions.add_aliases ["-no-results-function"]

let () = Parameter_customize.set_group performance
module ResultsAll =
  True

```

```

    (struct
      let option_name = "-eva-results"
      let help = "record values for any of the statements of the program."
    end)
let () = add_dep ResultsAll.parameter
let () = ResultsAll.add_aliases ["-results"]

let () = Parameter_customize.set_group performance
module JoinResults =
  Bool
  (struct
    let option_name = "-eva-join-results"
    let help = "precompute consolidated states once value is computed"
    let default = true
  end)
let () = JoinResults.add_aliases ["-val-join-results"]

let () = Parameter_customize.set_group performance
module EqualityStorage =
  Bool
  (struct
    let option_name = "-eva-equality-storage"
    let help = "Stores the states of the equality domain during \
      the analysis."
    let default = true
  end)
let () = add_precision_dep EqualityStorage.parameter

let () = Parameter_customize.set_group performance
module SymbolicLocsStorage =
  Bool
  (struct
    let option_name = "-eva-symbolic-locations-storage"
    let help = "Stores the states of the symbolic locations domain during \
      the analysis."
    let default = true
  end)
let () = add_precision_dep SymbolicLocsStorage.parameter

let () = Parameter_customize.set_group performance
module GaugesStorage =
  Bool
  (struct
    let option_name = "-eva-gauges-storage"
    let help = "Stores the states of the gauges domain during the analysis."
    let default = true
  end)
let () = add_precision_dep GaugesStorage.parameter

let () = Parameter_customize.set_group performance
module ApronStorage =
  Bool
  (struct
    let option_name = "-eva-apron-storage"
    let help = "Stores the states of the apron domains during the \
      analysis."
    let default = false
  end)
let () = add_precision_dep ApronStorage.parameter

let () = Parameter_customize.set_group performance
module BitwiseOffsmStorage =

```



```

Bool
  (struct
    let option_name = "-eva-bitwise-storage"
    let help = "Stores the states of the bitwise domain during the \
              analysis."
    let default = true
  end)
let () = add_precision_dep BitwiseOffsmStorage.parameter

(* ----- *)
(* --- Non-standard alarms --- *)
(* ----- *)

let () = Parameter_customize.set_group alarms
module AllRoundingModesConstants =
  False
  (struct
    let option_name = "-eva-all-rounding-modes-constants"
    let help = "Take into account the possibility of constants not being
converted to the nearest representable value, or being converted to higher
precision"
  end)
let () = add_correctness_dep AllRoundingModesConstants.parameter
let () = AllRoundingModesConstants.add_aliases ["-all-rounding-modes-constants"]

let () = Parameter_customize.set_group alarms
module UndefinedPointerComparisonPropagateAll =
  False
  (struct
    let option_name = "-eva-undefined-pointer-comparison-propagate-all"
    let help = "if the target program appears to contain undefined pointer
comparisons, propagate both outcomes {0; 1} in addition to the emission of an
alarm"
  end)
let () = add_correctness_dep UndefinedPointerComparisonPropagateAll.parameter
let () =
  UndefinedPointerComparisonPropagateAll.add_aliases
  ["-undefined-pointer-comparison-propagate-all"]

let () = Parameter_customize.set_group alarms
module WarnPointerComparison =
  String
  (struct
    let option_name = "-eva-warn-undefined-pointer-comparison"
    let help = "warn on all pointer comparisons, on comparisons where \
              the arguments have pointer type (default), or never warn"
    let default = "pointer"
    let arg_name = "all|pointer|none"
  end)
let () = WarnPointerComparison.set_possible_values ["all"; "pointer"; "none"]
let () = add_correctness_dep WarnPointerComparison.parameter
let () = WarnPointerComparison.add_aliases ["-val-warn-undefined-pointer-
comparison"]

let () = Parameter_customize.set_group alarms
let () = Parameter_customize.is_invisible ()
module WarnLeftShiftNegative =
  True
  (struct
    let option_name = "-val-warn-left-shift-negative"
    let help =

```

```

    "Emit alarms when left-shifting negative integers"
end)
let () = add_correctness_dep WarnLeftShiftNegative.parameter
let () = WarnLeftShiftNegative.add_update_hook
(fun _ v ->
  warning "This option is deprecated. Use %s instead"
    Kernel.LeftShiftNegative.name;
  Kernel.LeftShiftNegative.set v)

let () = Parameter_customize.set_group alarms
module WarnSignedConvertedDowncast =
  False
  (struct
    let option_name = "-eva-warn-signed-converted-downcast"
    let help = "Signed downcasts are decomposed into two operations: \
      a conversion to the signed type of the original width, \
      then a downcast. Warn when the downcast may exceed the \
      destination range."
  end)
let () = add_correctness_dep WarnSignedConvertedDowncast.parameter
let () =
  WarnSignedConvertedDowncast.add_aliases
    ["-val-warn-signed-converted-downcast"]

let () = Parameter_customize.set_group alarms
module WarnPointerSubstraction =
  True
  (struct
    let option_name = "-eva-warn-pointer-substraction"
    let help =
      "Warn when subtracting two pointers that may not be in the same \
      allocated block, and return the pointwise difference between the \
      offsets. When unset, do not warn but generate imprecise offsets."
  end)
let () = add_correctness_dep WarnPointerSubstraction.parameter
let () = WarnPointerSubstraction.add_aliases ["-val-warn-pointer-substraction"]

let () = Parameter_customize.set_group alarms
module IgnoreRecursiveCalls =
  False
  (struct
    let option_name = "-eva-ignore-recursive-calls"
    let help =
      "Pretend function calls that would be recursive do not happen. Causes
      unsoundness"
  end)
let () = add_correctness_dep IgnoreRecursiveCalls.parameter
let () = IgnoreRecursiveCalls.add_aliases ["-val-ignore-recursive-calls"]

let () = Parameter_customize.set_group alarms

module WarnCopyIndeterminate =
  Kernel_function_set
  (struct
    let option_name = "-eva-warn-copy-indeterminate"
    let arg_name = "f | @all"
    let help = "warn when a statement of the specified functions copies a \
      value that may be indeterminate (uninitialized or containing
      escaping address). \
      Set by default; can be deactivated for function 'f' by '=f',
      or for all \

```

```

        functions by '--@all'."
    end)
let () = add_correctness_dep WarnCopyIndeterminate.parameter
let () = WarnCopyIndeterminate.add_aliases ["-val-warn-copy-indeterminate"]
let () = WarnCopyIndeterminate.Category.(set_default (all ()))

let () = Parameter_customize.set_group alarms
module ReduceOnLogicAlarms =
    False
    (struct
        let option_name = "-eva-reduce-on-logic-alarms"
        let help = "Force reductions by a predicate to ignore logic alarms \
            emitted while the predicated is evaluated (experimental)"
    end)
let () = add_correctness_dep ReduceOnLogicAlarms.parameter
let () = ReduceOnLogicAlarms.add_aliases ["-val-reduce-on-logic-alarms"]

let () = Parameter_customize.set_group alarms
module InitializedLocals =
    False
    (struct
        let option_name = "-eva-initialized-locals"
        let help = "Local variables enter in scope fully initialized. \
            Only useful for the analysis of programs buggy w.r.t. \
            initialization."
    end)
let () = add_correctness_dep InitializedLocals.parameter
let () = InitializedLocals.add_aliases ["-val-initialized-locals"]

(* ----- *)
(* --- Initial context                               --- *)
(* ----- *)

let () = Parameter_customize.set_group initial_context
module AutomaticContextMaxDepth =
    Int
    (struct
        let option_name = "-eva-context-depth"
        let default = 2
        let arg_name = "n"
        let help = "use <n> as the depth of the default context for Eva. (defaults
to 2)"
    end)
let () = add_correctness_dep AutomaticContextMaxDepth.parameter
let () = AutomaticContextMaxDepth.add_aliases ["-context-depth"]

let () = Parameter_customize.set_group initial_context
module AutomaticContextMaxWidth =
    Int
    (struct
        let option_name = "-eva-context-width"
        let default = 2
        let arg_name = "n"
        let help = "use <n> as the width of the default context for Eva. (defaults
to 2)"
    end)
let () = AutomaticContextMaxWidth.set_range ~min:1 ~max:max_int
let () = add_correctness_dep AutomaticContextMaxWidth.parameter
let () = AutomaticContextMaxWidth.add_aliases ["-context-width"]

let () = Parameter_customize.set_group initial_context
module AllocatedContextValid =

```

```

False
  (struct
    let option_name = "-eva-context-valid-pointers"
    let help = "only allocate valid pointers until context-depth, and then use
NULL (defaults to false)"
  end)
let () = add_correctness_dep AllocatedContextValid.parameter
let () = AllocatedContextValid.add_aliases ["-context-valid-pointers"]

let () = Parameter_customize.set_group initial_context
module InitializationPaddingGlobals =
  String
  (struct
    let default = "yes"
    let option_name = "-eva-initialization-padding-globals"
    let arg_name = "yes|no|maybe"
    let help = "Specify how padding bits are initialized inside global \
initialized), \
                <no> (padding is completely uninitialized), or <maybe> \
                (padding may be uninitialized). Default is <yes>."
  end)
let () = InitializationPaddingGlobals.set_possible_values ["yes"; "no"; "maybe"]
let () = add_correctness_dep InitializationPaddingGlobals.parameter
let () = InitializationPaddingGlobals.add_aliases ["-val-initialization-padding-
globals"]

(* ----- *)
(* --- Tuning --- *)
(* ----- *)

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.is_invisible ()
module DescendingIteration =
  String
  (struct
    let default = "no"
    let option_name = "-eva-descending-iteration"
    let arg_name = "no|exits|full"
    let help = "Experimental. After hitting a postfix point, try to improve \
the precision with either a <full> iteration or an iteration
from loop \
                head to exit paths (<exits>) or do not try anything (<no>).
Default \
                is <no>."
  end)
let () = DescendingIteration.set_possible_values ["no" ; "exits" ; "full"]
let () = add_precision_dep DescendingIteration.parameter

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.is_invisible ()
module HierarchicalConvergence =
  False
  (struct
    let option_name = "-eva-hierarchical-convergence"
    let help = "Experimental and unsound. Separate the convergence process \
convergence of \
                of each levels of nested loops. This implies that the
iteration \
                inner loops will be completely recomputed when doing another
                of the outer loops."
  end)

```

```

let () = add_precision_dep HierarchicalConvergence.parameter

let () = Parameter_customize.set_group precision_tuning
module WideningDelay =
  Int
  (struct
    let default = 3
    let option_name = "--eva-widening-delay"
    let arg_name = "n"
    let help =
      "do not widen before the <n>-th iteration (defaults to 3)"
  end)
let () = WideningDelay.set_range ~min:1 ~max:max_int
let () = WideningDelay.add_aliases ["-wlevel"]
let () = add_precision_dep WideningDelay.parameter

let () = Parameter_customize.set_group precision_tuning
module WideningPeriod =
  Int
  (struct
    let default = 2
    let option_name = "--eva-widening-period"
    let arg_name = "n"
    let help =
      "after the first widening, widen each <n> iterations (defaults to 2)"
  end)
let () = WideningDelay.set_range ~min:1 ~max:max_int
let () = add_precision_dep WideningPeriod.parameter

let () = Parameter_customize.set_group precision_tuning
module ILevel =
  Int
  (struct
    let option_name = "--eva-ilevel"
    let default = 8
    let arg_name = "n"
    let help =
      "Sets of integers are represented as sets up to <n> elements. \
      Above, intervals with congruence information are used \
      (defaults to 8, must be between 4 and 128)"
  end)
let () = add_precision_dep ILevel.parameter
let () = ILevel.add_aliases ["-val-ilevel"]
let () = ILevel.add_update_hook (fun _ i -> Ival.set_small_cardinal i)
let () = ILevel.set_range 4 128

let () = Parameter_customize.set_group precision_tuning
module SemanticUnrollingLevel =
  Zero
  (struct
    let option_name = "--eva-slevel"
    let arg_name = "n"
    let help =
      "superpose up to <n> states when unrolling control flow. The larger n,
      the more precise and expensive the analysis (defaults to 0)"
  end)
let () = add_precision_dep SemanticUnrollingLevel.parameter
let () = SemanticUnrollingLevel.add_aliases ["-slevel"]

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.argument_may_be_fundecl ()
module SlevelFunction =

```

```

Kernel_function_map
(struct
  include Datatype.Int
  type key = Cil_types.kernel_function
  let of_string ~key:_ ~prev:_ s =
    Extlib.opt_map
      (fun s ->
        try int_of_string s
        with Failure _ ->
          raise (Cannot_build ("'" ^ s ^ "' is not an integer")))
      s
  let to_string ~key:_ = Extlib.opt_map string_of_int
end)
(struct
  let option_name = "-eva-slevel-function"
  let arg_name = "f:n"
  let help = "override slevel with <n> when analyzing <f>"
  let default = Kernel_function.Map.empty
end)
let () = add_precision_dep SlevelFunction.parameter
let () = SlevelFunction.add_aliases ["-slevel-function"]

let () = Parameter_customize.set_group precision_tuning
module SlevelMergeAfterLoop =
  Kernel_function_set
  (struct
    let option_name = "-eva-slevel-merge-after-loop"
    let arg_name = "f | @all"
    let help =
      "when set, the different execution paths that originate from the body \
      of a loop are merged before entering the next excution."
  end)
let () = add_precision_dep SlevelMergeAfterLoop.parameter
let () = SlevelMergeAfterLoop.add_aliases ["-val-slevel-merge-after-loop"]

let () = Parameter_customize.set_group precision_tuning
module MinLoopUnroll =
  Int
  (struct
    let option_name = "-eva-min-loop-unroll"
    let arg_name = "n"
    let default = 0
    let help =
      "unroll <n> loop iterations for each loop, regardless of the slevel \
      settings and the number of states already propagated. \
      Can be overwritten on a case by case basis by loop unroll annotations."
  end)
let () = add_precision_dep MinLoopUnroll.parameter
let () = MinLoopUnroll.set_range 0 max_int

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.argument_may_be_fundecl ()
module SplitReturnFunction =
  Kernel_function_map
  (struct
    (* this type is ad-hoc: cannot use Kernel_function_multiple_map here *)
    include Split_strategy
    type key = Cil_types.kernel_function
    let of_string ~key:_ ~prev:_ s =
      try Extlib.opt_map Split_strategy.of_string s
      with Split_strategy.ParseFailure s ->
        raise (Cannot_build ("unknown split strategy " ^ s))
  end)

```

```

    let to_string ~key:_ v =
      Extlib.opt_map Split_strategy.to_string v
    end)
  (struct
    let option_name = "-eva-split-return-function"
    let arg_name = "f:n"
    let help = "split return states of function <f> according to \
      \\result == n and \\result != n"
    let default = Kernel_function.Map.empty
  end)
let () = add_precision_dep SplitReturnFunction.parameter
let () = SplitReturnFunction.add_aliases ["-val-split-return-function"]

let () = Parameter_customize.set_group precision_tuning
module SplitReturn =
  String
  (struct
    let option_name = "-eva-split-return"
    let arg_name = "mode"
    let default = ""
    let help = "when 'mode' is a number, or 'full', this is equivalent \
      to -val-split-return-function f:mode for all functions f. \
      When mode is 'auto', automatically split states at the end \
      of all functions, according to the function return code"
  end)
module SplitGlobalStrategy = State_builder.Ref (Split_strategy)
  (struct
    let default () = Split_strategy.NoSplit
    let name = "Value_parameters.SplitGlobalStrategy"
    let dependencies = [SplitReturn.self]
  end)
let () =
  SplitReturn.add_set_hook
  (fun _ x -> SplitGlobalStrategy.set
    (try Split_strategy.of_string x
      with Split_strategy.ParseFailure s ->
        abort "@[@[incorrect argument for option %s@ (%s).@]"
          SplitReturn.name s))
let () = add_precision_dep SplitReturn.parameter
let () = SplitReturn.add_aliases ["-val-split-return"]

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.argument_may_be_fundec1 ()
module BuiltinsOverrides =
  Kernel_function_map
  (struct
    include Datatype.String
    type key = Cil_types.kernel_function
    let of_string ~key:kf ~prev:_ nameopt =
      begin match nameopt with
      | Some name ->
        if not (!Db.Value.mem_builtin name) then
          abort "option '-val-builtin %a:%s': undeclared builtin '%s'@.\
            declared builtins: @[%a@]"
            Kernel_function.pretty kf name name
            (Pretty_utils.pp_list ~sep:",,@ " Format.pp_print_string)
            (List.map fst (!Db.Value.registered_builtins ()))
        | _ -> abort
          "option '-val-builtin':@ \
            no builtin associated to function '%a',@ use '%a:<builtin>'"
            Kernel_function.pretty kf Kernel_function.pretty kf
      end;
  end);

```

```

    nameopt
    let to_string ~key:_ name = name
end)
(struct
  let option_name = "-eva-builtin"
  let arg_name = "f:ffc"
  let help = "when analyzing function <f>, try to use Frama-C builtin \
    <ffc> instead. \
    Fall back to <f> if <ffc> cannot handle its arguments."
  let default = Kernel_function.Map.empty
end)
let () = add_precision_dep BuiltinsOverrides.parameter
let () = BuiltinsOverrides.add_aliases ["-val-builtin"]

let () = Parameter_customize.set_group precision_tuning
module BuiltinsAuto =
  True
  (struct
    let option_name = "-eva-builtins-auto"
    let help = "When set, builtins will be used automatically to replace \
      known C functions"
  end)
let () = add_correctness_dep BuiltinsAuto.parameter
let () = BuiltinsAuto.add_aliases ["-val-builtins-auto"]

let () = Parameter_customize.set_group precision_tuning
module BuiltinsList =
  False
  (struct
    let option_name = "-eva-builtins-list"
    let help = "Lists the existing builtins, and which functions they \
      are automatically associated to (if any)"
  end)
let () = BuiltinsList.add_aliases ["-val-builtins-list"]

let () = Parameter_customize.set_group precision_tuning
module LinearLevel =
  Zero
  (struct
    let option_name = "-eva-subdivide-non-linear"
    let arg_name = "n"
    let help =
      "Improve precision when evaluating expressions in which a variable \
        appears multiple times, by splitting its value at most n times. \
        Defaults to 0."
  end)
let () = add_precision_dep LinearLevel.parameter
let () = LinearLevel.add_aliases ["-val-subdivide-non-linear"]

let () = Parameter_customize.set_group precision_tuning
let () = Parameter_customize.argument_may_be_fundecl ()
module UsePrototype =
  Kernel_function_set
  (struct
    let option_name = "-eva-use-spec"
    let arg_name = "f1,..,fn"
    let help = "use the ACSL specification of the functions instead of their
definitions"
  end)
let () = add_precision_dep UsePrototype.parameter
let () = UsePrototype.add_aliases ["-val-use-spec"]

```



```

let () = Parameter_customize.set_group precision_tuning
module SkipLibcSpecs =
  True
  (struct
    let option_name = "-eva-skip-stdlib-specs"
    let help = "skip ACSL specifications on functions originating from the \
      standard library of Frama-C, when their bodies are evaluated"
  end)
let () = add_precision_dep SkipLibcSpecs.parameter
let () = SkipLibcSpecs.add_aliases ["-val-skip-stdlib-specs"]

let () = Parameter_customize.set_group precision_tuning
module RmAssert =
  True
  (struct
    let option_name = "-eva-remove-redundant-alarms"
    let help = "after the analysis, try to remove redundant alarms, so that
the user needs inspect fewer of them"
  end)
let () = add_precision_dep RmAssert.parameter
let () = RmAssert.add_aliases ["-remove-redundant-alarms"]

let () = Parameter_customize.set_group precision_tuning
module MemExecAll =
  True
  (struct
    let option_name = "-eva-memexec"
    let help = "Speed up analysis by not recomputing functions already \
      analyzed in the same context. Forces -inout-callwise. \
      Callstacks for which the analysis has not been recomputed \
      are incorrectly shown as dead in the GUI."
  end)
let () = MemExecAll.add_aliases ["-memexec-all"]
let () =
  MemExecAll.add_set_hook
  (fun _ bold bnew ->
    if bnew then
      try
        Dynamic.Parameter.Bool.set "-inout-callwise" true
        with Dynamic.Unbound_value _ | Dynamic.Incompatible_type _ ->
          abort "Cannot set option -eva-memexec. Is plugin Inout registered?"
      )

let () = Parameter_customize.set_group precision_tuning
module ArrayPrecisionLevel =
  Int
  (struct
    let default = 200
    let option_name = "-eva-plevel"
    let arg_name = "n"
    let help = "use <n> as the precision level for arrays accesses. \
      Array accesses are precise as long as the interval for the
index contains \
      less than n values. (defaults to 200)"
  end)
let () = add_precision_dep ArrayPrecisionLevel.parameter
let () = ArrayPrecisionLevel.add_aliases ["-plevel"]
let () = ArrayPrecisionLevel.add_update_hook
  (fun _ v -> Offsetmap.set_plevel v)

(* Options SaveFunctionState and LoadFunctionState are related

```

```

and mutually dependent for sanity checking.
Also, they depend on BuiltinOverrides, so they cannot be defined before it.
*)
let () = Parameter_customize.set_group initial_context
module SaveFunctionState =
  Kernel_function_map
  (struct
    include Datatype.String
    type key = Cil_types.kernel_function
    let of_string ~key:_ ~prev:_ file = file
    let to_string ~key:_ file = file
  end)
  (struct
    let option_name = "-eva-save-fun-state"
    let arg_name = "function:filename"
    let help = "save state of function <function> in file <filename>"
    let default = Kernel_function.Map.empty
  end)
let () = SaveFunctionState.add_aliases ["-val-save-fun-state"]
let () = Parameter_customize.set_group initial_context
module LoadFunctionState =
  Kernel_function_map
  (struct
    include Datatype.String
    type key = Cil_types.kernel_function
    let of_string ~key:_ ~prev:_ file = file
    let to_string ~key:_ file = file
  end)
  (struct
    let option_name = "-eva-load-fun-state"
    let arg_name = "function:filename"
    let help = "load state of function <function> from file <filename>"
    let default = Kernel_function.Map.empty
  end)
let () = LoadFunctionState.add_aliases ["-val-load-fun-state"]
let () = add_correctness_dep SaveFunctionState.parameter
let () = add_correctness_dep LoadFunctionState.parameter
(* checks that SaveFunctionState has a unique argument pair, and returns it. *)
let get_SaveFunctionState () =
  let is_first = ref true in
  let (kf, filename) = SaveFunctionState.fold
    (fun (kf, opt_filename) _acc ->
      if !is_first then is_first := false
      else abort "option `%s' requires a single function:filename pair"
        SaveFunctionState.name;
      let filename = Extlib.the opt_filename in
      kf, filename
    ) (Kernel_function.dummy (), "")
  in
  if filename = "" then abort "option `%s' requires a function:filename pair"
    SaveFunctionState.name
  else kf, filename
(* checks that LoadFunctionState has a unique argument pair, and returns it. *)
let get_LoadFunctionState () =
  let is_first = ref true in
  let (kf, filename) = LoadFunctionState.fold
    (fun (kf, opt_filename) _acc ->
      if !is_first then is_first := false
      else abort "option `%s' requires a single function:filename pair"
        LoadFunctionState.name;
      let filename = Extlib.the opt_filename in
      kf, filename
    ) (Kernel_function.dummy (), "")
  in
  if filename = "" then abort "option `%s' requires a function:filename pair"
    LoadFunctionState.name
  else kf, filename

```

```

    ) (Kernel_function.dummy (), "")
in
if filename = "" then abort "option `s' requires a function:filename pair"
  LoadFunctionState.name
else kf, filename
(* perform early sanity checks to avoid aborting the analysis only at the end *)
let () = Ast.apply_after_computed (fun _ ->
  (* check the function to save returns 'void' *)
  if SaveFunctionState.is_set () then begin
    let (kf, _) = get_SaveFunctionState () in
    if not (Kernel_function.returns_void kf) then
      abort "option `s': function `%a' must return void"
        SaveFunctionState.name Kernel_function.pretty kf
    end;
  if SaveFunctionState.is_set () && LoadFunctionState.is_set () then begin
    (* check that if both save and load are set, they do not specify the
       same function name (note: cannot compare using function ids) *)
    let (save_kf, _) = get_SaveFunctionState () in
    let (load_kf, _) = get_LoadFunctionState () in
    if Kernel_function.equal save_kf load_kf then
      abort "options `s' and `s' cannot save/load the same function `%a'"
        SaveFunctionState.name LoadFunctionState.name
        Kernel_function.pretty save_kf
    end;
  if LoadFunctionState.is_set () then
    let (kf, _) = get_LoadFunctionState () in
    BuiltinOverrides.add (kf, Some "Frama_C_load_state");
  )
)

(* ----- *)
(* --- Messages --- *)
(* ----- *)

let () = Parameter_customize.set_group messages
module ValShowProgress =
  False
  (struct
    let option_name = "-eva-show-progress"
    let help = "Show progression messages during analysis"
  end)
let () = ValShowProgress.add_aliases ["-val-show-progress"]

let () = Parameter_customize.set_group messages
let () = Parameter_customize.is_invisible ()
module ValShowInitialState =
  True
  (struct
    let option_name = "-val-show-initial-state"
    (* deprecated in Silicon *)
    let help = "[deprecated] Show initial state before analysis starts. \
               This option has been replaced by \
               -value-msg-key=[-]initial-state and has no effect anymore."
  end)
let () =
  ValShowInitialState.add_set_hook
  (fun _ new_ ->
    if new_ then
      Kernel.warning "@[Option -val-show-initial-state has no effect, \
                    it has been replaced by -eva-msg-key=initial-state@]"
    else
      Kernel.warning "@[Option -no-val-show-initial-state has no effect, \
                    it has been replaced by -eva-msg-key=-initial-state@]"
  )

```

```

)

let () = Parameter_customize.set_group messages
module ValShowPerf =
  False
  (struct
    let option_name = "-eva-show-perf"
    let help = "Compute and shows a summary of the time spent analyzing
function calls"
  end)
let () = ValShowPerf.add_aliases ["-val-show-perf"]

let () = Parameter_customize.set_group messages
module ValPerfFlamegraphs =
  String
  (struct
    let option_name = "-eva-flamegraph"
    let help = "Dumps a summary of the time spent analyzing function calls \
in a format suitable for the Flamegraph tool \
(http://www.brendangregg.com/flamegraphs.html)"
    let arg_name = "file"
    let default = ""
  end)
let () = ValPerfFlamegraphs.add_aliases ["-val-flamegraph"]

let () = Parameter_customize.set_group messages
module ShowSlevel =
  Int
  (struct
    let option_name = "-eva-show-slevel"
    let default = 100
    let arg_name = "n"
    let help = "Period for showing consumption of the allotted slevel during
analysis"
  end)
let () = ShowSlevel.add_aliases ["-val-show-slevel"]
let () = ShowSlevel.set_range ~min:1 ~max:max_int

let () = Parameter_customize.set_group messages
module PrintCallstacks =
  False
  (struct
    let option_name = "-eva-print-callstacks"
    let help = "When printing a message, also show the current call stack"
  end)
let () = PrintCallstacks.add_aliases ["-val-print-callstacks"]

let () = Parameter_customize.set_group messages
let () = Parameter_customize.is_invisible ()
module AlarmsWarnings =
  True
  (struct
    let option_name = "-val-warn-on-alarms"
    let help = "[DEPRECATED: use warning key alarm to manage alarms] \
if set (default), possible alarms are printed in \
the analysis log as warnings, otherwise as plain feedback"
  end)

let () =
  AlarmsWarnings.add_set_hook
  (fun _ f ->

```

```

match get_warn_status wkey_alarm with
| Log.Wabort | Log.Werror | Log.Werror_once ->
  warning "alarms already set to produce an error. \
          Ignoring -val-warn-on-alarms"
| Log.Winactive | Log.Wactive | Log.Wfeedback ->
  set_warn_status wkey_alarm (if f then Log.Wactive else Log.Wfeedback)
| Log.Wonce | Log.Wfeedback_once ->
  (* Keep the 'once' status. Note that this will only happen if user
     is mixing old and new style of warning management, thus it becomes
     difficult to interpret the desired action.
  *)
  set_warn_status wkey_alarm
    (if f then Log.Wonce else Log.Wfeedback_once)

let () = Parameter_customize.set_group messages
module ReportRedStatuses =
  String
  (struct
    let option_name = "-eva-report-red-statuses"
    let arg_name = "filename"
    let default = ""
    let help = "output the list of \"red properties\" in a csv file of the \
              given name. These are the properties which were invalid for \
              some states. Their consolidated status may not be invalid, \
              but they should often be investigated first."
  end)

let () = Parameter_customize.set_group messages
module NumerorsLogFile =
  String
  (struct
    let option_name = "-eva-numerors-log-file"
    let help = "The Numerors Domain will save each call to the DPRINT \
              function in the given file"
    let arg_name = "file"
    let default = ""
  end)

let () = Parameter_customize.set_group alarms
let () = Parameter_customize.is_invisible ()
module WarnBuiltinOverride =
  True(struct
    let option_name = "-val-warn-builtin-override"
    let help = "[DEPRECATED: use warning category key '\" ^
              (wkey_name wkey_builtins_override) ^
              '\" to control] Warn when Eva built-ins will override function \
              definitions"
  end)
let () = add_correctness_dep WarnBuiltinOverride.parameter
let () = WarnBuiltinOverride.add_update_hook
  (fun _ v ->
    warning "Option %s is deprecated. \
            Use warning category key '%a' instead"
            WarnBuiltinOverride.option_name
            pp_warn_category wkey_builtins_override;
    set_warn_status wkey_builtins_override
      (if v then Log.Wonce else Log.Winactive))

(* ----- *)
(* --- Interpreter mode --- *)
(* ----- *)

```

```

let () = Parameter_customize.set_group interpreter
module InterpreterMode =
  False
  (struct
    let option_name = "-eva-interpreter-mode"
    let help = "Stop at first call to a library function, if main() has \
      arguments, on undecided branches"
  end)
let () = InterpreterMode.add_aliases ["-val-interpreter-mode"]

let () = Parameter_customize.set_group interpreter
let () = Parameter_customize.is_invisible ()
module ObviouslyTerminatesFunctions =
  Fundec_set
  (struct
    let option_name = "-obviously-terminates-function"
    let arg_name = "f"
    let help = "deprecated"
  end)
let () = add_dep ObviouslyTerminatesFunctions.parameter
let () = ObviouslyTerminatesFunctions.add_update_hook
(fun _ _ ->
  warning "Option -obviously-terminates-function is no longer supported. \
    Ignoring.")

let () = Parameter_customize.set_group interpreter
let () = Parameter_customize.is_invisible ()
module ObviouslyTerminatesAll =
  False
  (struct
    let option_name = "-obviously-terminates"
    let help = "undocumented and deprecated"
  end)
let () = add_dep ObviouslyTerminatesAll.parameter
let () = ObviouslyTerminatesAll.add_update_hook
(fun _ _ ->
  warning "Option -obviously-terminates is no longer supported. \
    Ignoring.")

let () = Parameter_customize.set_group interpreter
module StopAtNthAlarm =
  Int(struct
    let option_name = "-eva-stop-at-nth-alarm"
    let default = max_int
    let arg_name = "n"
    let help = "Aborts the analysis when the nth alarm is emitted."
  end)
let () = StopAtNthAlarm.add_aliases ["-val-stop-at-nth-alarm"]

(* ----- *)
(* --- Ugliness required for correctness --- *)
(* ----- *)

let () = Parameter_customize.is_invisible ()
module InitialStateChanged =
  Int (struct
    let option_name = "-eva-new-initial-state"
    let default = 0
    let arg_name = "n"
    let help = ""
  end)
(* Changing the user-supplied initial state (or the arguments of main) through

```

the API of Db.Value does reset the state of Value, but **not** the property statuses that Value has positioned. Currently, statuses can only depend on a command-line parameter. We use the dummy one above to force a reset when needed. *)

```

let () =
  add_correctness_dep InitialStateChanged.parameter;
  Db.Value.initial_state_changed :=
    (fun () -> InitialStateChanged.set (InitialStateChanged.get () + 1))

(* ----- *)
(* --- Eva options --- *)
(* ----- *)

let () = Parameter_customize.set_group precision_tuning
module EnumerateCond =
  Bool
  (struct
    let option_name = "-eva-enumerate-cond"
    let help = "Activate reduce_by_cond_enumerate."
    let default = true
  end)
let () = add_precision_dep EnumerateCond.parameter

let () = Parameter_customize.set_group precision_tuning
module OracleDepth =
  Int
  (struct
    let option_name = "-eva-oracle-depth"
    let help = "Maximum number of successive uses of the oracle by the domain \
\
    for the evaluation of an expression. Set 0 to disable the \
    oracle."
    let default = 2
    let arg_name = ""
  end)
let () = add_precision_dep OracleDepth.parameter

let () = Parameter_customize.set_group precision_tuning
module ReductionDepth =
  Int
  (struct
    let option_name = "-eva-reduction-depth"
    let help = "Maximum number of successive backward reductions that the \
    domain may initiate."
    let default = 4
    let arg_name = ""
  end)
let () = add_precision_dep ReductionDepth.parameter

(* ----- *)
(* --- Dynamic allocation --- *)
(* ----- *)

let () = Parameter_customize.set_group malloc
module MallocFunctions=
  Filled_string_set
  (struct
    let option_name = "-eva-malloc-functions"
    let arg_name = "f1,...,fn"
  end)

```

```

    let help = "The malloc builtins use the call site of malloc() to know \
              where to create new bases. This detection does not work for \
              custom allocators or wrappers on top of malloc, unless they \
              are listed here. By default, only contains malloc."
    let default = Datatype.String.Set.singleton "malloc"
  end)
let () = MallocFunctions.add_aliases ["-val-malloc-functions"]

let () = Parameter_customize.set_group malloc
module AllocReturnsNull=
  True
  (struct
    let option_name = "-eva-alloc-returns-null"
    let help = "Memory allocation built-ins (malloc, calloc, realloc) are \
              modeled as nondeterministically returning a null pointer"
  end)
let () = AllocReturnsNull.add_aliases ["-val-alloc-returns-null"]

let () = Parameter_customize.set_group malloc
module MallocLevel =
  Int
  (struct
    let option_name = "-eva-mlevel"
    let default = 0
    let arg_name = "m"
    let help = "sets to [m] the number of precise dynamic allocation for any \
              given callstack"
  end)
let () = MallocLevel.add_aliases ["-val-mlevel"]

(* ----- *)
(* --- Freeze parameters. MUST GO LAST --- *)
(* ----- *)

let parameters_correctness =
  Typed_parameter.Set.elements !parameters_correctness
let parameters_tuning =
  Typed_parameter.Set.elements !parameters_tuning
let parameters_abstractions =
  Typed_parameter.Set.elements !parameters_abstractions

(*
Local Variables:
compile-command: "make -C ../../.."
End:
*)

```

- File compute_functions.ml

```

(*****)
(*                                           *)
(* This file is part of Frama-C.           *)
(*                                           *)
(* Copyright (C) 2007-2018                 *)
(* CEA (Commissariat à l'énergie atomique et aux énergies *)
(* alternatives)                          *)
(*                                           *)
(* you can redistribute it and/or modify it under the terms of the GNU *)
(* Lesser General Public License as published by the Free Software *)

```



```

(* Foundation, version 2.1. *)
(* *)
(* It is distributed in the hope that it will be useful, *)
(* but WITHOUT ANY WARRANTY; without even the implied warranty of *)
(* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the *)
(* GNU Lesser General Public License for more details. *)
(* *)
(* See the GNU Lesser General Public License version 2.1 *)
(* for more details (enclosed in the file licenses/LGPLv2.1). *)
(* *)
(*****)

open Cil_types
open Eval

let dkey = Value_parameters.register_category "callbacks"

let floats_ok () =
  let u = min_float /. 2. in
  let u = u /. 2. in
  assert (0. < u && u < min_float)

let need_assigns kf =
  let spec = Annotations.funspec ~populate:false kf in
  match Cil.find_default_behavior spec with
  | None -> true
  | Some bhv -> bhv.b_assigns = WritesAny

let options_ok () =
  (* Check that we can parse the values specified for the options that require
     advanced parsing. Just make a query, as this will force the kernel to
     parse them. *)
  let check f = try ignore (f ()) with Not_found -> () in
  check Value_parameters.SplitReturnFunction.get;
  check Value_parameters.BuiltinsOverrides.get;
  check Value_parameters.SlevelFunction.get;
  check Value_parameters.EqualityCallFunction.get;
  let check_assigns kf =
    if need_assigns kf then
      Value_parameters.error "[no assigns@ specified@ for function '%a',@ for \
used.@ \
which@ a builtin@ or the specification@ will be
Potential unsoundness.@]" Kernel_function.pretty
kf
  in
  Value_parameters.BuiltinsOverrides.iter (fun (kf, _) -> check_assigns kf);
  Value_parameters.UsePrototype.iter (fun kf -> check_assigns kf)

(* Do something tasteless in case the user did not put a spec on functions
   for which he set [-val-use-spec]: generate an incorrect one ourselves *)
let generate_specs () =
  let aux kf =
    if need_assigns kf then begin
      let spec = Annotations.funspec ~populate:false kf in
      Value_parameters.warning "Generating potentially incorrect assigns \
for function '%a' for which option %s is set"
Kernel_function.pretty kf Value_parameters.UsePrototype.option_name;
      (* The function populate_spec may emit a warning. Position a loc. *)
      Cil.CurrentLoc.set (Kernel_function.get_location kf);
      ignore (!Annotations.populate_spec_ref kf spec)
    end
  in

```

```

Value_parameters.UsePrototype.iter aux

let pre_analysis () =
  floats_ok ();
  options_ok ();
  Split_return.pretty_strategies ();
  generate_specs ();
  Widen.precompute_widen_hints ();
  if Value_parameters.WarnBuiltinOverride.get () then
    Builtins.warn_definitions_overridden_by_builtins ();
  Value_perf.reset ();
  (* We may be resuming Value from a previously crashed analysis. Clear
     degeneration states *)
  Value_util.DegenerationPoints.clear ();
  Cvalue.V.clear_garbled_mix ();
  Value_util.clear_call_stack ();
  Db.Value.mark_as_computed ()

let post_analysis_cleanup ~aborted =
  Value_util.clear_call_stack ();
  (* Precompute consolidated states if required *)
  if Value_parameters.JoinResults.get () then
    Db.Value.Table_By_Callstack.iter
      (fun s _ -> ignore (Db.Value.get_stmt_state s));
  if not aborted then begin
    (* Keep memexec results for users that want to resume the analysis *)
    Mem_exec.cleanup_results ();
    if not (Value_parameters.SaveFunctionState.is_empty ()) then
      State_import.save_globals_state ();
  end

let post_analysis () =
  (* Garbled mix must be dumped here -- at least before the call to
     mark_green_and_red -- because fresh ones are created when re-evaluating
     all the alarms, and we get an unpleasant "ghost effect". *)
  Value_util.dump_garbled_mix ();
  (* Mark unreachable and RTE statuses. Only do this there, not when the
     analysis was aborted (hence, not in post_cleanup), because the
     propagation is incomplete. Also do not mark unreachable statutes if
     there is an alarm in the initializers (bottom initial state), as we
     would end up marking the alarm as dead. *)
  Eval_annots.mark_unreachable ();
  (* Try to refine the 'Unknown' statuses that have been emitted during
     this analysis. *)
  Eval_annots.mark_green_and_red ();
  Eval_annots.mark_rte ();
  post_analysis_cleanup ~aborted:false;
  (* Remove redundant alarms *)
  if Value_parameters.RmAssert.get () then !Db.Value.rm_asserts ()

(* Register a signal handler for SIGUSR1, that will be used to abort Value *)
let () =
  let prev = ref (fun _ -> ()) in
  let handler (_signal: int) =
    !prev Sys.sigusr1; (* Call previous signal handler *)
    Value_parameters.warning "Stopping analysis at user request@";
    Partitioned_dataflow.signal_abort ()
  in
  try
    match Sys.signal Sys.sigusr1 (Sys.Signal_handle handler) with
    | Sys.Signal_default | Sys.Signal_ignore -> ()
    | Sys.Signal_handle f -> prev := f

```

```

with Invalid_argument _ -> () (* Ignore: SIGURSR1 is not available on Windows,
                               and possibly on other platforms. *)
module Make
  (Abstract: Abstractions.S)
  (Eva: Evaluation.S with type value = Abstract.Val.t
    and type origin = Abstract.Dom.origin
    and type loc = Abstract.Loc.location
    and type state = Abstract.Dom.t)
= struct

  module Domain = struct
    include Abstract.Dom
    let enter_scope kf vars state = match vars with
      | [] -> state
      | _ -> enter_scope kf vars state
    let leave_scope kf vars state = match vars with
      | [] -> state
      | _ -> leave_scope kf vars state
  end
  module PowersetDomain = Powerset.Make (Domain)

  module Transfer =
    Transfer_stmt.Make (Abstract.Val) (Abstract.Loc) (Domain) (Eva)

  module Logic = Transfer_logic.Make (Domain) (PowersetDomain)

  module Spec =
    Transfer_specification.Make
      (Abstract.Val) (Abstract.Loc) (Domain) (PowersetDomain) (Logic)

  module Init = Initialization.Make (Abstract.Dom) (Eva) (Transfer)

  module Computer =
    Partitioned_dataflow.Computer
      (Domain) (PowersetDomain) (Transfer) (Init) (Logic) (Spec)

  let initial_state = Init.initial_state

  let get_cvalue =
    match Domain.get Cvalue_domain.key with
    | None -> fun _ -> Cvalue.Model.top
    | Some get -> fun state -> get state

  let get_cval =
    match Abstract.Val.get Main_values.cvalue_key with
    | None -> fun _ -> assert false
    | Some get -> fun value -> get value

  let get_ploc =
    match Abstract.Loc.get Main_locations.ploc_key with
    | None -> fun _ -> assert false
    | Some get -> fun location -> get location

  (* Compute a call to [kf] in the state [state]. The evaluation will
     be done either using the body of [kf] or its specification, depending
     on whether the body exists and on option [-val-use-spec]. [call_kinstr]
     is the instruction at which the call takes place, and is used to update
     the statuses of the preconditions of [kf]. If [show_progress] is true,
     the callstack and additional information are printed. *)
  let compute_using_spec_or_body call_kinstr call state =
    (* ABP add the following block *)
    begin

```

```

    let kf_name = Kernel_function.get_name call.kf in
    if (String.equal kf_name "gets") || (String.equal kf_name "toto") then
        Value_parameters.warning ~once:true ~current:true
~wkey:Value_parameters.wkey_dangerous_functions "calling dangerous function'%a"
Kernel_function.pretty call.kf
    end;
    let kf = call.kf in
    Value_results.mark_kf_as_called kf;
    let global = match call_kinstr with Kglobal -> true | _ -> false in
    let pp = not global && Value_parameters.ValShowProgress.get () in
    let call_stack = Value_util.call_stack () in
    if pp then
        Value_parameters.feedback
            "@[computing for function %a.@\nCalled from %a.@]"
            Value_types.Callstack.pretty_short call_stack
            Cil_datatype.Location.pretty (Cil_datatype.Kinstr.loc call_kinstr);
    let use_spec =
        if call.recursive then
            `Spec (Recursion.empty_spec_for_recursive_call kf)
        else
            match kf.fundec with
            | Declaration (_,_,_,_) -> `Spec (Annotations.funspec kf)
            | Definition (def, _) ->
                if Kernel_function.Set.mem kf (Value_parameters.UsePrototype.get ())
                then `Spec (Annotations.funspec kf)
                else `Def def
    in
    let cvalue_state = get_cvalue state in
    let resulting_states, cacheable = match use_spec with
    | `Spec spec ->
        Db.Value.Call_Type_Value_Callbacks.apply
            (`Spec spec, cvalue_state, call_stack);
        if Value_parameters.InterpreterMode.get ()
        then Value_parameters.abort "Library function call. Stopping.";
        Value_parameters.feedback ~once:true
            "@[using specification for function %a@]" Kernel_function.pretty kf;
        let vi = Kernel_function.get_vi kf in
        if Cil.hasAttribute "fc_stdlib" vi.vattr then
            Library_functions.warn_unsupported_spec vi.vorig_name;
        Spec.compute_using_specification ~warn:true call_kinstr call spec state,
        Value_types.Cacheable
    | `Def fundec ->
        Db.Value.Call_Type_Value_Callbacks.apply (`Def, cvalue_state,
call_stack);
        Computer.compute kf call_kinstr state
    in
    if pp then
        Value_parameters.feedback
            "Done for function %a" Kernel_function.pretty kf;
    Transfer.{ states = resulting_states; cacheable; builtin=false }

(* Mem Exec *)

module MemExec = Mem_exec.Make (Abstract.Val) (Domain)

let compute_and_cache_call stmt call init_state =
    let default () = compute_using_spec_or_body (Kstmt stmt) call init_state in
    if Value_parameters.MemExecAll.get () then
        let args =
            List.map (fun {avalue} -> Eval.value_assigned avalue) call.arguments
        in

```

```

match MemExec.reuse_previous_call call.kf init_state args with
| None ->
  let call_result = default () in
  let () =
    if not (!Db.Value.use_spec_instead_of_definition call.kf)
      && call_result.Transfer.cacheable = Value_types.Cacheable
    then
      let final_states = call_result.Transfer.states in
      MemExec.store_computed_call call.kf init_state args final_states
  in
  call_result
| Some (states, i) ->
  let stack_with_call = Value_util.call_stack () in
  Db.Value.Call_Type_Value_Callbacks.apply
    (`Memexec, get_cvalue init_state, stack_with_call);
  (* Evaluate the preconditions of kf, to update the statuses
     at this call. *)
  let spec = Annotations.funspec call.kf in
  if not (Value_util.skip_specifications call.kf) &&
    Eval_annots.has_requires spec
  then begin
    let ab = Logic.create_init_state call.kf in
    ignore (Logic.check_fct_preconditions
      (Kstmt stmt) call.kf ab init_state);
  end;
  if Value_parameters.ValShowProgress.get () then begin
    Value_parameters.feedback ~current:true
      "Reusing old results for call to %a" Kernel_function.pretty call.kf;
    Value_parameters.debug ~dkey
      "calling Record_Value_New callbacks on saved previous result";
  end;
  let stack_with_call = Value_util.call_stack () in
  Db.Value.Record_Value_Callbacks_New.apply
    (stack_with_call, Value_types.Reuse i);
  (* call can be cached since it was cached once *)
  Transfer.{states; cacheable = Value_types.Cacheable; builtin=false}
else
  default ()

let get_cvalue_call call =
  let lift_left left = { left with lloc = get_ploc left.lloc } in
  let lift_flagged_value value = { value with v = value.v >>-: get_cval } in
  let lift_assigned = function
    | Assign value -> Assign (get_cval value)
    | Copy (lval, value) -> Copy (lift_left lval, lift_flagged_value value)
  in
  let lift_argument arg = { arg with avalue = lift_assigned arg.avalue } in
  let arguments = List.map lift_argument call.arguments in
  let rest = List.map (fun (e, assgn) -> e, lift_assigned assgn) call.rest in
  { call with arguments; rest }

let join_states = function
| [] -> `Bottom
| [state] -> `Value state
| s :: l -> `Value (List.fold_left Domain.join s l)

let compute_call_or_builtin stmt call state =
  match Builtins.find_builtin_override call.kf with
  | None -> compute_and_cache_call stmt call state
  | Some (name, builtin, spec) ->
    Value_results.mark_kf_as_called call.kf;
    let kinstr = Kstmt stmt in

```

```

let kf_name = Kernel_function.get_name call.kf in
if Value_parameters.ValShowProgress.get ()
then
  Value_parameters.feedback ~current:true "Call to builtin %s%s"
    name (if kf_name = name then "" else " for function " ^ kf_name);
  (* Do not track garbled mixes created when interpreting the specification,
    as the result of the cvalue builtin will overwrite them. *)
  Locations.Location_Bytes.do_track_garbled_mix false;
  let states =
    Spec.compute_using_specification ~warn:false kinstr call spec state
  in
  Locations.Location_Bytes.do_track_garbled_mix true;
  let final_state = states >>- join_states in
  let cvalue_state = get_cvalue state in
  match final_state with
  | `Bottom ->
    let cs = Value_util.call_stack () in
    Db.Value.Call_Type_Value_Callbacks.apply (`Spec spec, cvalue_state, cs);
    let cacheable = Value_types.Cacheable in
    Transfer.{states; cacheable; builtin=true}
  | `Value final_state ->
    let cvalue_call = get_cvalue_call call in
    let cvalue_states, cacheable =
      Builtins.apply_builtin builtin cvalue_call cvalue_state
    in
    let insert (cvalue_state, clobbered_set) =
      Domain.set Locals_scoping.key clobbered_set
        (Domain.set Cvalue_domain.key cvalue_state final_state)
    in
    let states = Bottom.bot_of_list (List.map insert cvalue_states) in
    Transfer.{states; cacheable; builtin=true}

let compute_call =
  if Domain.mem Cvalue_domain.key
  && Abstract.Val.mem Main_values.cvalue_key
  && Abstract.Loc.mem Main_locations.ploc_key
  then compute_call_or_builtin
  else compute_and_cache_call

let () = Transfer.compute_call_ref := compute_call

let store_initial_state kf init_state =
  Domain.Store.register_initial_state (Value_util.call_stack ()) init_state;
  let cvalue_state = get_cvalue init_state in
  Db.Value.Call_Value_Callbacks.apply (cvalue_state, [kf, Kglobal])

let compute kf init_state =
  try
    Value_util.push_call_stack kf Kglobal;
    store_initial_state kf init_state;
    let call =
      {kf; arguments = []; rest = []; return = None; recursive = false}
    in
    let final_result = compute_using_spec_or_body Kglobal call init_state in
    let final_states = final_result.Transfer.states in
    let final_state = PowersetDomain.(final_states >>-: of_list >>- join) in
    Value_util.pop_call_stack ();
    Value_parameters.feedback "done for function %a" Kernel_function.pretty
kf;
    post_analysis ();
    Domain.post_analysis final_state;
  with
with

```

```

| Db.Value.Aborted ->
post_analysis_cleanup ~aborted:true;
(* Signal that a degeneration occurred *)
if Value_util.DegenerationPoints.length () > 0 then
  Value_parameters.error
  "Degeneration occurred:@\nresults are not correct for lines of code \
  that can be reached from the degeneration point.@."

let compute_from_entry_point kf ~lib_entry =
  pre_analysis ();
  Value_parameters.feedback "Analyzing a%complete application starting at %a"
    (if lib_entry then "n in" else " ")
  Kernel_function.pretty kf;
  let initial_state =
    try Init.initial_state_with_formals ~lib_entry kf
    with Db.Value.Aborted ->
      post_analysis_cleanup ~aborted:true;
      Value_parameters.abort "Degeneration occurred during initialization,
aborting."
  in
  match initial_state with
  | `Bottom ->
    Value_parameters.result "Eva not started because globals \
    initialization is not computable.";
    Eval_annots.mark_invalid_initializers ()
  | `Value init_state ->
    compute kf init_state

let compute_from_init_state kf init_state =
  pre_analysis ();
  Domain.Store.register_global_state (`Value init_state);
  compute kf init_state
end

(*
Local Variables:
compile-command: "make -C ../../../../.."
End:
*)

```