TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technology

ITC70LT

Daniele Mucci – 156440IVCM

TED - THE ELF DOCTOR A CONTAINER BASED TOOL TO PERFORM SECURITY RISK ASSESSMENT FOR ELF BINARIES

Master's Thesis

Supervisor: Bernhards Blumbergs
MSc
TUT Cyber security PhD student

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

ITC70LT

Daniele Mucci – 156440IVCM

TED - THE ELF DOCTOR
KONTEINERIPÕHINE TÖÖRIIST HINDAMAKS TURVARISKE ELF PROGRAMMIDES

Magistritöö

Juhendaja: Bernhards Blumbergs
MSc
TUT Küberkaitse doktorant

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and this thesis has not been presented for examination or submitted for defence anywhere else. All used materials, references to the literature and work of others have been cited.

Author: Daniele Mucci

07.05.2018

Abstract

This thesis presents TED, a novel tool, based on containers, that performs security risk assessment on binaries for Linux systems. This project starts from building the threat landscape for binaries to later analyze the currently available defense measures which can be used to protect ELF files from exploitation and reverse engineering. The study finds a noticeable discrepancy between the defenses produced in the academic context and the ones used in the industry, which are in general older and often do not offer protection against the most recently developed attacks. Despite this, TED aims to be a tool usable in real-world scenarios, therefore it includes the most commonly deployed measures and it is tested in a wide range of environments, from simple virtual machines to real servers and complex infrastructures. The results of the tests performed show that TED is able to effectively provide an overview of the risk associated with binaries in a system, and also that the choice of basing the application on Docker containers is very beneficial, allowing to greatly reduce the dependencies and to easily integrate different tools, and entails a negligible performance penalization.

Keywords: Risk assessment, Linux, ELF binaries, Docker, containers.

This thesis is in English and contains 76 pages of text, 7 chapters, 6 figures, 26 tables.

Annotatsioon

TED - The ELF Doctor

Konteineripõhine tööriist hindamaks turvariske ELF programmides

Käesolevas töös tutvustatakse TED-i, innovaatilist tööriista, mis põhineb kapsuleeritud konteinerite tehnoloogial ning mis teostab Linuxi operatsioonisüsteemide binaarfailide turvariski hindamist. Töö esimeses pooles identifitseeritakse binaarfailidele suunatud ohte, millest edasi jõutakse praeguste kaitsemeetmete analüüsini, mida saab rakendada ELF failide kaitsmiseks ekspluateerimise ja pöördprojekteerimise eest. Töö käigus ilmnes, et akadeemilises kontekstis ja tarkvaratehnikas kasutatakse iganenud kaitsemeetodeid, mis ei paku piisavalt kaitset nüüdisaegsetele rünnakute ja -meetodite vastu. Vaatamata sellele, saab TED-i juba kasutada igapäeva töös, kuna tööriist hõlmab endas modernseid kaitsepraktikaid ning turvameetmeid, mida on testitud erinevates keskkondades, alustades lihtsamatest virtuaalmasinatest serveriparkideni. Töös läbiviidud testide tulemus näitab, et TED suudab anda informatiivse ülevaate süsteemi binaarfailidele suunatud ohuvektoritest ning tõestab Dockeri konteinerite arhitektuuri kasulikkust. See võimaldab suuresti vähendada kolmandate osapoolte sõltuvusi ning kergevaevaliselt lisada pistikprogramme, millel on kaduvväikene mõju jõudlusele.

Märksõnad: Riskianalüüs, Linux, ELF binaarfailid, Docker, konteinerid

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 76 leheküljel, 7 peatükki, 6 joonist, 26 tabelit.

List of abbreviations and terms

AAAS	ASCII Armored Address Space
AES	Advanced Encryption Standard
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
AVX	Advanced Vector Extension
CLI	Command Line Interface
CPU	Central Processing Unit
CR	Carriage Return
CSIRT	Computer Security Incident Response Team
CSV	Comma Separated Value
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GID	Group ID
GOT	Global Offset Table
HTTP	HyperText Transfer Protocol
IPC	Interprocess Communications
IT	Information and Technology
JSON	JavaScript Object Notation
KAISER	Kernel Address Isolation to have Side-channels Efficently Removed
KASLR	Kernel Address Space Layout Randomization
KPTI	Kernel Page Table Isolation
LF	Line Feed
LIFO	Last In First Out
LSB	Least Significant Bit
LXC	Linux containers
LXD	Linux container hypervisor
NATO	North Atlantic Treaty Organization
NIS	Network Information Service
NOP	No Operation
NX	No eXecute
PID	Process ID

ROP	Return Oriented Programming
SHA256	Secure Hasing Algorithm, 256-Bits
SMAP	Supervisor Mode Access Prevention
SMEP	Supervisor Mode Execution Prevention
SSH	Secure Shell
SSP	Stack Smashing Protector
STDERR	Standard Error
STDOUT	Standard Output
TED	The Elf Doctor
TCP	Transmission Control Protocol
UID	User ID
UPX	Ultimate Packer for eXecutables
USB	Universal Serial Bus
UTS	UNIX Time Sharing
W^X	Writable XOR Executable
XOR	Exclusive OR

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Research Questions

This project is developed keeping in mind the following research questions:

1. What are the possibilities -how and with which tools- to assess possible attacks against binaries in the Linux ecosystem or to individuate possible vulnerable executables in an already compromised server?

2. How the existing tools, aimed to assess or detect such attacks, can be improved by the use of containerization? What are the benefits of such approach?

1.2 Motivations for this project and personal contributions

It is no news anymore that cyber attacks have been increasing during the recent years, proportionally to the crucial role of IT (Information and Technology) systems and their growing number [59]. Despite many of these computer incidents are related to web applications, low level attacks like the ones described in this thesis are still present and dangerous. In October 2017 for example, a heap overflow vulnerability in the DNSAPI dynamic library in Windows has been discovered, allowing potential attackers remote code execution capabilities on vulnerable machines [65]. In the same month two heap-based buffer overflow vulnerabilities were discovered in the *wget* Linux utility, potentially allowing code execution to attackers [19]. Moreover, two out of three servers used for hosting websites or web applications are running a Linux distribution and in general Unix or Unix-like operating systems dominate the market when it comes to servers, mainframes, supercomputers or anything that is not a personal computer, including mobile phones [22], making the security of such systems of vital importance. Therefore, the idea behind this project is to offer a reliable, scalable, effective and portable tool to verify that the existing security measures for protecting binaries are in place, and to discover possible weak links for future attacks or insecure binaries that might have been exploited by attackers to compromise the machine. In order to do this, already existing tools will be integrated and eventually extended into a framework which will make extensive use of containers as execution environment.
Given this purpose, the novelty of this work is represented by the following points:

1. Evaluation of container engines in the perspective of vulnerability assessment and incident response. At the present day there are multiple ways to achieve containerization. In fact the use of containers, despite being an old technology, has seen a major increase only during the last few years (from 2015), and different alternatives have been and continue to be developed, in particular Docker and LXC. Given the requirements outlined, an evaluation of which container engines is the best fit for this project is necessary in order to establish the trade-off between the various platforms, taking into account functionality, dependencies, availability and performances.

2. Application of containers in the context of security assessment and incident response. This is innovative both from the methodological and from the technical perspective. The tools aimed to perform binary analysis are in general quite old and in many cases require some specific dependency to be satisfied in order to run. This can consume precious time in case of incident response and can also potentially change and therefore pollute the system in object, not considering possible problems of compatibility. The use of containers strongly mitigates the dependency issue and therefore speeds up

the deployment of such tools. Moreover, this project aims to integrate a set of tools that otherwise should be deployed individually, collecting the individual results into a global overview.

3. Production of a prototype of the tool whose characteristics have been outlined in the previous points. This implementation will be done mostly in Python, C and bash, with extensive use of the most appropriate container engine. The idea is to provide a utility that uses a specific set of tools, runs inside one or more containers, tests all or a meaningful part of the binaries present on the machine, collects the results and finally performs an assessment of the machine in exam. Since containers share the kernel with the host they run on, it is possible to perform also some checks concerning the security measures applied to the kernel.

4. As an addition to the previous points, it might be necessary to modify small parts or simply update some of the tools used to fit into the use case of this project or to develop custom supporting tools. Despite this further development might be necessary it is not to be intended as the main objective of this project but just as a complementary work functional to achieve the main goals.

1.3 Structure of the thesis

This thesis is structured as follows:
Section 2 presents a brief introduction on the purpose and target audience of this project. Then, section 3 describes the subject of the project and provides the literature review. Here the general characteristics of Linux binaries and how they are executed and organized is outlined in order to create a solid background to understand the threats against them and the corresponding mitigations. In addition to describing the ELF format, a summary of the most common and the most relevant attacks is presented; topics such as buffer overflow and heap overflow attacks are discussed, together with reverse engineering and novel threats such as Spectre and Meltdown. Because this project is not focused on attacks, these will be discussed with the sole purpose of highlighting the importance of their countermeasures and to better explain their working mechanisms. In this same section the existing defense mechanisms and mitigation techniques will be discussed, analyzing what is the current state of the art in this matter. This is the theoretical core of the thesis, since the main purpose of TED is exactly to detect whether these measures are in place or have been applied.
The final theoretical discussion in section 3.7 discusses the common characteristics of containers.

Before going into the details of how this project has been developed, in section 4 the methodology used will be presented, describing the high-level process adopted to conduct the research, from the design to the validation of the results.
In section 5 the practical core of this thesis will be presented. In this section the details concerning the implementation of the tool, ranging from the choice of the container engine to the selection of the defense mechanisms to verify are discussed.
Section 6 contains the description of the testing and validation process.
Lastly, the conclusions drawn from the development of the project, the future work and the fields in which more research is needed are presented in section 7.

2 Scope and motivation of the tool

2.1 General purpose of the tool

TED aims to protect computers by identifying possible vulnerabilities before an actual incident happens (proactive approach) or to help investigating an incident that has already happened (analytical approach).
In both the cases mentioned, TED focuses on auditing the system and analyzing the current state, without taking any action directly. In this perspective, TED does not take into account the reason why a given binary is in the system or why a certain defense measure is or is not present, leaving to the specialist performing the auditing the responsibility to determine whether each executable is legitimate, malicious, vulnerable or whether it might have been exploited. The tool developed works on UNIX systems with a very limited set of dependencies. Because TED makes large use of containers to run, it faces the constraint imposed by this technology about sharing the kernel with the host it runs on. The proactive approach consists mainly in scanning all the binaries that are on the machine and analyzing which ones make use of defense techniques and which do not, making the latter more vulnerable to attacks such as those described in section 3.5.
The analytical approach instead consists in using this tool as an investigation tool to detect possible compromised or vulnerable binaries, giving better insights and helping the team who is performing the investigation to respond to an incident. For this reason, the use of this tool is specifically appropriate for teams or individuals preventing or investigating security incidents on UNIX machines, such as CSIRTs (Computer Security Incident Response Teams), digital forensic investigators or system administrators.

2.2 CSIRTs as addressees

CSIRTs, Computer Security Incident Response Teams respond to a need for companies and institutions that is growing over time. Despite the security measures that might be in place, sooner or later many constituencies occur in a violation of their system or in a computer incident. Because of this, it is needed for organizations to identify with rapidity and accuracy that the problem occurred and to respond to it [74]. CSIRTs provide this functionality by supporting a given organization to prevent, detect and respond to computer security incidents. The services provided by a CSIRT include, but are not limited to, reporting incidents, defining a threat environment for the organization, detecting actual malicious or suspicious events, prioritizing and analyzing security incidents and responding to them [74]. Despite most CSIRT started as "response-oriented", it is common for them to focus on prevention of incidents and also on the learning process after an incident to avoid it from happening again [55]. TED can fit in the context of CSIRTs by providing a summarized view of the systems in object, gathering information that can help reducing the time spent on general investigations and pointing the investigators in the direction of possible vulnerabilities. Moreover, by gathering both system and file specific data, TED makes it easier to correlate information and perform more accurate guesses.

2.3 Digital Forensic teams as addressees

The term "Digital-Forensic" indicates a set of proven effective methods to preserve, collect, validate, identify, analyze, interpret, document and present digital evidences [63]. The focus of forensic teams is therefore on the evidence represented by data on the machine in object. In order not to tamper the eventual evidence, forensic investigators will never work on the original data but only on a clone of this [63]. This practice is what makes the tool presented suitable in a digital forensic investigation scenario. After a certain device has been seized, the drives of that machine will be cloned and on those clones the analysis will follow to understand if and what evidence is present. The tool presented can therefore support the analysis and ease the collection of evidences by identifying potential security vulnerabilities in the machine, when this is the victim of an attack or a security breach. In the case of digital forensic, this tool would have exclusively an analytical approach. It is important to note that TED might modify the content of the drive, specifically because of the need of a container engine, that might not be already installed. This tampering can be avoided, but in general contradicts one of the four principles of digital evidence [69] and must be kept into careful consideration.

2.4 System Administrators as addressees

TED, as discussed, can be used in two different perspectives: proactive and analytical. This means that those who have the responsibility over a UNIX system can find its capabilities useful in order to prevent attacks or to trace back a possible attack vector. It is a common and good practice for system administrators to enforce proactive measures on the systems they manage, and TED can be used as a proactive tool to have a better knowledge of all the binaries present on the system, of the security of those binaries and of the global likelihood that their system might be compromised through an attack against them.

3 Background information and literature review

In this section the foundations to build this project will be laid. First, in section 3.2 the files which represent the main target of this project will be analyzed. Then, an overview of the architecture on which the research is made will be presented in section 3.3, together with its main features. Once the operating environment for the project has been outlined, the threat landscape will be presented in section 3.5 and the available defense tools and techniques will be discussed in section 3.6. Finally, section 3.7 outlines the main characteristics of containers.

3.1 Related work

Extensive research have been performed on binary security, both from the attack and from the defense perspective, as discussed in section 3.5 and 3.6. However, very few projects with a purpose similar to the one presented in this thesis were found. A comparable project is from 2008, called BitBlaze and developed by Song et al. [79], which unlike TED uses both static and dynamic analysis to extract security information from the program, without taking in consideration the defense measures applied and relying on custom techniques. The main purpose of BitBlaze is therefore to detect vulnerabilities in the program, rather than determining what security measures are applied. Young-Hyun et al. [31] in 2015 developed a project with the aim of dynamically detecting vulnerabilities in binaries with taint analysis. This project therefore focuses on finding vulnerabilities or detecting exploitation at runtime, whereas TED does not execute any binary and simply verifies whether proper defenses are in place. A similar approach has been followed by Ulrich F. [84] who presents TEASER, a system which aims to assess the exploitability of binaries, performing therefore a vulnerability assessment from the perspective of an attacker. TANG et al. [42] and Wang et al. [87] focused on binary security analysis in terms of performing a diagnosis of memory vulnerabilities and on tracing back an exploit to its root, respectively. Both these project have a very different perspective and use case, compared to TED. Outside the academic research, a project that can compared with TED is *checksec.sh*[1] which is a shell script that shows which security measures are applied to a binary or to the kernel. Despite TED has some similarities with *checksec*, the two projects have a different scope (checksec focuses on a single binary or on the kernel, whereas TED aims to give an overview of the whole system), purpose (checksec simply displays technical information, without performing an evaluation of the risks associated with the information gathered) and make use of different technologies, with TED being much more portable and easier to extend.

From the analyzed literature, no projects aiming to perform binary risk assessment from the

[1]`https://github.com/slimm609/checksec.sh`

defense perspective in a fashion similar to TED were found, and it is therefore possible to conclude that TED uses both a novel approach and a new technology in the field of binary security.

3.2 Binary Files

This project focuses on the security of Linux servers and specifically on assessing the risk associated with binary files, it is therefore important to describe the main structure and features of these files. In this section first an overview of the ELF format will be presented, while in the following sections the most important parts of binary files will be discussed in more detail.

3.2.1 General Overview.

The word *Binaries* in a Unix environment usually refers to a specific format, called Executable Linkable Format (ELF). This format is used not only for executables but also for shared libraries, core dumps and object files [40, p.9]. There are three main types of ELF files:

Relocatable files. These are often called simply *object files*, they are composed of position independent code suitable for linking with other object files in order to create an executable [40, p.10]. These files have usually a *.o* extension.

Executable files. These are files suitable for execution, often called simply *programs* and they represent the entry point for a process [34, p.1-1].

Shared Objects. These are files that hold both code and data, they are often called *shared libraries*[34, p1-1]. Shared object files can be linked with other relocatable or shared objects and then dynamically linked to create a process image.

The name Executable Linkable Format comes from the fact that these files offer parallel views for the same content, both from the linking and from the execution perspective.
As it is possible to see in Figure 1, the content of the file can be variable. For example, an object file, that is not meant to be executed, might not have a Program Header Table while an executable file might not have a Section Header Table. Independently from the type, an ELF header is always present, as it holds the most important meta-information about the file [34, p1-2].

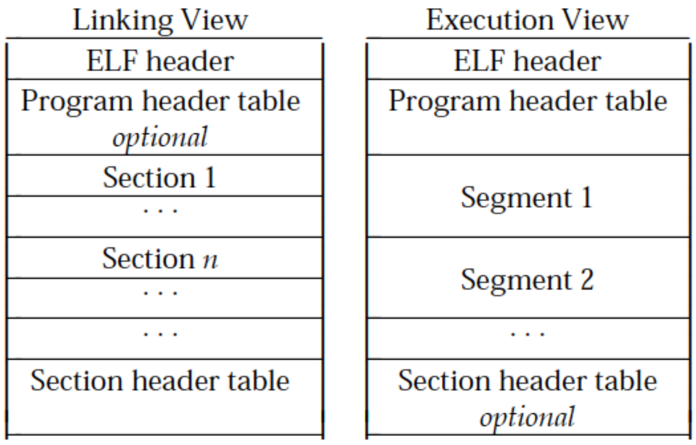


Figure 1. The parallel views for an ELF file. Taken from [34, p.1-1]

3.2.2 ELF header

The ELF header is a *struct* that holds various information about the file that represents. The main elements contained in the ELF header are reported in Table 2 [34, p.1-4,1-8]. In the context of this project it is important to note that the first field, *e_ident* contains a magic number that is used to identify the file as an ELF.

3.2.3 Program headers

For the executable view, it is necessary that the ELF file contains the Program Header Table. The entries in this table are used to describe the segments within the binary.
Each entry carries information about a specific segment. The most important information is in the *p_type* field. Here it can be specified if the segment is of type PT_LOAD, meaning that the segment is going to be loaded in memory (for example the Text segment), PT_NAME, meaning that the segment contains the location of a path name for the interpreter to use, PT_DYNAMIC for dynamically linked executables and so on [34, p.2-2,2-4]. Each program header table entry contains also information about the size and location of the segment, its memory alignment requirements and the flags relevant for the segment; these flags include Read, Write and Execute, establishing what permissions should be assigned to each segment [34, p.2-2].

3.2.4 Section headers

Similarly to the program headers, the section headers are the entries of the section header table and describe the sections present in the binary. It is important to remember that the

Table 2. *The main fields in the ELF header and their function.*

Name	**Description**
e_ident	Identification of the ELF, including a magic number, specification of 32bit or 64bit, data encoding and version.
e_type	Type of the ELF file. This can be Relocatable, Executable, Shared-object, Core dump and so on.
e_machine	Architecture required, a possible value can be 3 for i386, 7 for 8086 etc. .
e_entry	Virtual address of the entry point from where to start the execution, if applicable, otherwise 0.
e_phoff	Offset in the file to find the Program Header Table.
e_shoff	Offset in the file to find the Section Header Table.
e_flags	Processor specific flags.
e_ehsize	Size of the ELF header.
e_phentsize	Size of one entry of the Program Header Table.
e_shentsize	Size of one entry of the Section Header Table.
e_phnum	Number of entries in the Program Header Table.
e_shnum	Number of entries in the Section Header Table.

sections, unlike segments, are not necessary for the execution of a binary, as these are mainly used for linking and debugging [40, p.16-17].
The information contained in an entry of the section header table includes the name of the section it refers to and its type, flags, and finally size, location and alignment details [34, p.1-9,1-14].
There are many different types of sections, including symbol or string table, symbol hash table and relocation entries. Some of the sections will be discussed in more detail in section 3.4, especially the ones commonly subject to exploitation.

3.3 Architecture of the systems in object

Many topics discussed in this thesis involve characteristics of the systems which are "low level" and therefore related with the architecture of the target machine. Because of this, before diving into the details concerning memory and processes, it is important to clarify which architecture this project targets. The choice in this context was between x86 and AMD64, which are respectively 32 and 64 bit architectures. A very common approach while dealing with topics that will also be covered in the following sections is to use x86 systems, which makes the examples simpler and makes it easier to compare older references whose authors, at the time of their writing, had not the possibility to choose. Despite it would be reasonable to also base this project on x86 architectures, the AMD64 architecture has been selected. The reason for this choice is that this project aims to produce a tool useful in the real world, where the absolute majority of systems are using 64bit processors. The following

Table 3. AMD64 main registers names and descriptions.

Register	Description
%rsp	stack pointer
%rbp	Callee-save, stack frame pointer
%r12-15	Callee-save
%rbx	Callee-save
%rdi	Caller-save, 1st argument
%rsi	Caller-save, 2nd argument
%rdx	Caller-save, 3rd argument
%rcx	Caller-save, 4th argument
%r8	Caller-save, 5th argument
%r9	Caller-save, 6th argument
%rax	Caller-save, return register
%r11	Caller-save

two sections will then deal briefly with the characteristics of AMD64 architectures such as registers used and calling convention, both very useful in order to remove ambiguity when memory layouts, processes and attacks will be described.

3.3.1 Registers in AMD64 architectures

Processors based on AMD64 architecture offer a bigger number of registers compared to 32-bit architectures; here in fact there are 16 general purpose registers 64-bit long, which are, together with their usage, the most interesting part to describe in this section. In this architecture there are also a number of 128-bit registers and 80-bit floating point registers. Most modern processors may contain also 16 256-bit registers (Intel AVX) or 32 512-bit registers (Intel AVX-512), but discussing this is not in the scope of this thesis [52, p.17]. A summary of the most interesting registers and what they are used for can be found in Table 3. A more in-depth description about the purpose of each register will be presented in section 3.3.2.

3.3.2 Calling convention

The calling convention describes how functions are called, how arguments are passed to them and how results are collected. It is interesting to present the calling convention in AMD64 architectures since this influences the memory layout of the processes. In fact, in addition to registers, each function has a frame on the system stack. A more in-depth discussion about this will be presented in section 3.4.1. When a function needs to be called, its arguments are first computed. After this operation is completed, the arguments are either placed in the registers, as described in Table 3 or placed on the stack. This is one of the main differences between AMD64 and x86 architectures. In the latter, in fact, all arguments were always pushed to the stack, independently from their length or format. In AMD64 systems instead, the arguments are first classified and then a decision whether to put them in the registers or to push them on the stack is taken according to this classification. There are several categories [52, p.19] but the most interesting ones are INTEGER and MEMORY. The first includes all integral types that are able to fit into one general purpose register, such as *_Bool*, *char*, *short*, *int*, *long*, *long long* and pointers. An argument is instead classified as type MEMORY if it is larger than 8 bytes or if it contains unaligned fields [52, p.18-21]. After the classification is made, arguments of type MEMORY are pushed to the stack, while arguments of type INTEGER are placed in the first free register among (in order) %rdi, %rsi, %rdx, %rcx, %r8 and %r9 [52, p.22]. It is worth mentioning that the order in which the arguments of type MEMORY are pushed on the stack follows the C convention [43, p.18] meaning that the arguments are pushed in reverse order making it easier to handle functions with variable number of arguments. In practical terms, the C convention implies that the first parameter of a function has the lowest address, right above the return address, while the last parameter is closer to the bottom of the stack, at a higher address [43, p.18]. Finally, the last specification described in the function calling convention concerns how functions should provide their results to the caller. This mechanism also depends on the classes specified above: if the type of the return is MEMORY, then the caller uses %rdi to provide to the callee the address at which the result must be written, as a sort of hidden argument. At the end of the execution, this same address will be in %rax. If the type is INTEGER, then the return value is stored in the first available register among %rax and %rdx.

3.4 Memory segmentation

After having presented some architecture-dependent elements about process execution, it is possible to examine how the memory is organized for a process, since this organization will be the subject of the exploitation techniques described in section 3.5 and partly of the defenses described in section 3.6.
In Unix environments, each process has its own dedicated portion of memory which is used to store the information needed by the process to run. As discussed in section 3.2.3 and 3.2.4, there are several segments in the memory of a program and each of them can be divided into multiple sections. The following overview describe briefly the most relevant memory sections, their use and their characteristics for a user level process.

Text. The text section, called *.text* holds the actual code of the program. This section is read-only and has a fixed size defined when the program is first loaded [41, p.37]. This sections is part of the Text segment [40].

Data. The data section, called *.data* is used to store the global or the static local variables that have been initialized. As for the previous section, its size is fixed at runtime [41, p.37]. Unlike *.text* though, this section is writable [44, p.5]. Because of the requirement to be writable, this section is part of the Data segment [40].

Below Stack Section. The below stack section, called *.bss* is complementary to the data section, since it holds the global or the local static variables that have not been initialized or that have been initialized with the default value -0 for integers and NULL for pointers. Even in this case, the size of this section is fixed at runtime and the section is writable [41, p.37] [44, p.5], therefore the *.bss* section is placed in the Data segment [40, p.19].

Initialization and destruction section. These sections, called *.init* and *.fini* respectively, are placed into the Text segment as they also hold executable instructions. Specifically, the instructions in the *.init* section will be executed before the main program entry point while the ones in *.fini* will be executed after it, if this exits normally [7].

Global Offset Table. The global offset table (GOT) is a section called *.got* or *.got.plt* that resides in the Data segment, since needs to be writable. The purpose of this table is to reference global variables or functions the belong to shared libraries [40, p.19]. The table is populated at process creation by the dynamic linker.

Environment/Command line. The Environment or Arguments section is used to store some environmental variables as well as command line arguments to the program itself. Examples for the system-level variables which are made available in this section are the hostname, the PATH and the shell name. This section is located at the highest addresses [41, p.38].

3.4.1 The Stack

The memory section known as stack supports a very important mechanism used in programs: the function. Using functions allows to reference and execute section of codes by name, just passing them the required parameters. The stack is nothing more than a LIFO (Last In First Out) queue, in which the last element added is the first to get out. From this derives that the

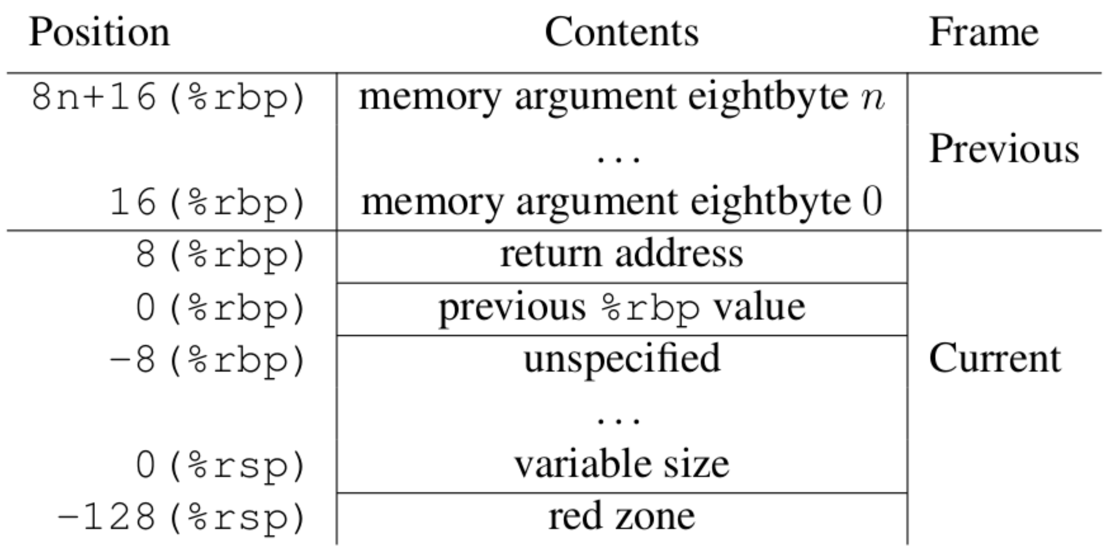
two most important operation defined on the stack are PUSH, used to insert elements on top of the stack, and POP, that removes elements from the top. It is important to note that the stack grows "backwards": the base is at higher addresses and each element is added under that, at lower addresses. Moreover, the stack address in AMD64 systems is always aligned on a 16 byte boundary [52, p.19].
In modern CPUs (Central Processing Units) there are dedicated instructions that implement PUSH and POP [20], and there are also specific registers to manage the stack. There is a stack pointer register (%rsp) that points to the end of the last allocated stack frame [52, p.18] and a stack frame pointer, called %rbp which, as seen in section 3.3.1, is used to setup the stack frame for a specific function during the prologue and epilogue, as it will be discussed in the next section. Moreover, the 128 bytes under the %rsp form the so-called *red zone*. This area can be used for temporary data that is not needed across function calls and it might be used to contain the entire stack frame of leaf functions, which are those that do not call other functions inside their bodies [52, p.19].

Stack and functions. In order to implement functions, for each call to function a section of the stack is reserved, called *stack frame* [17]. The simple program displayed in Figure 2 is used as an example for the description below. The *main* function is the first to be executed,

```
void function(int a, int b) {
    int array[5];
}
void main(){
    function(1,2);
    printf("Done");
}
```

Figure 2. Explanatory code to present the stack function.

despite it has its own record in the stack it complicates the explanation, and therefore will not be discussed. The *main* function contains a call to *function()*. This instruction breaks the flow of the program, since the next instruction that needs to be executed is inside the *function()* body. In order to support this execution, first the arguments to *function()*, in this case *1* and *2*, are moved into %rdi and %rsi respectively, then the return address is saved, which means that the address of the instruction that needs to be executed once the call to *function()* returns is pushed into the stack; in this case this is the address of the *printf* instruction. As soon as *function()* is called, it executes what is called the *prologue* [44, p.16]. This consists of saving (pushing) the old value of %rbp and copying the current value of %rsp into %rbp, so that the



Position	Contents	Frame
8n+16(%rbp)	memory argument eightbyte n	Previous
	...	
16(%rbp)	memory argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
−8(%rbp)	unspecified	
	...	
0(%rsp)	variable size	
−128(%rsp)	red zone	

Figure 3. A summary view of a stack frame for a given function call. Taken from [52, p.18]

latter can be used to reference local addresses to the stack. After the prologue is completed, the only instruction of *function()* is executed, allocating space for the array. Local variables which are not initialized use the stack, so allocating space for this array means subtracting 5x4 bytes to the stack pointer. This case shows one more time the use of %rbp, that can be used as a reference point unlike %rsp that instead can change, for example to allocate space for local variables. Now that all the instructions of the very simple *function()* have been executed, what is usually called *epilogue* is executed, reversing what has been done in the prologue. What happens in the epilogue is that the stack pointer is assigned back to the current frame pointer, freeing the local variable space, and the old %rbp value is popped and restored. After this, the return address is popped and inserted into the program counter so that the execution can continue, in this case to the *printf* instruction. For both prologue and epilogue there are built-in instructions in modern systems (*enter* and *leave*) and also in 64-bit systems the prologue is not always present in this form; it is in fact possible to compute during compilation the space that a given function call will need to execute. Because of this, it is possible to find compiled code that does not save the previous %rbp and instead simply subtracts from the stack pointer the amount needed -usually all at once- and then in the epilogue adds back to the stack pointer the same amount, reverting the state as it was before the function call [37]. This kind of practice of avoiding the "standard" prologue can be done also by some compilers for a matter of optimization. In these cases the compiled code will use offsets to %rsp in order to reference local variables, even if this should change during the execution, since all this information is known at the time of compilation [44, p.552-555].

3.4.2 The Heap

The heap is a dynamic sections of memory that is used mostly for arrays and data structures. This memory section grows from lower addresses toward higher addresses. As for the stack, each process has its own heap. The most common way, but not the only one, to manage the memory on the heap is through *malloc()* and *free()*. These functions are not system calls, but just a wrapper around *brk()* and *sbrk()*, system calls that offer the possibility to get some contiguous memory locations at request. The memory allocated by *malloc* can be freed with a corresponding *free* call. If this does not happen, then the memory is wasted (leaked) and cannot be reused until the whole process terminates [28].
Because in the majority of the cases the heap is managed through *malloc/free*, a brief overview of how these functions work in the simplest implementation will be presented.

malloc(). The primary *malloc* mechanism is to allocate a large block of memory with *brk()*, which means moving the break boundary of the heap toward the limit by a certain amount. Once this portion of memory is available, there must be an algorithm in place in order to manage efficiently the memory allocation for the program. A basic concept in this environment is a chunk: *malloc* divides the portion of memory it can manage into chunks of different sizes. In order to be efficient, each chunk stores also some metadata. If the chunk is in use, its size is stored in the header, and this allows to find the beginning of the next chunk [10]. If the chunk is not in use, more information can be stored to facilitate other processes; this include a footer, which is a copy of the chunk size (except for the 3 LSB which are set to 0 unlike in the header, where they are used as flags). It is also possible for free chunks to contain pointers to other chunks and if they are big enough they also might contain the size of the next chunk. In the context of this project, it is mainly relevant to point out that each chunk contains a header that states its size, and this is used to navigate in one direction or the other across chunks or to free the memory, since this is one of the mechanisms that will be exploited in heap-based attacks and it is a feature that does not depend much from the specific *malloc()* implementation.

free(). The basic role of *free()* is to add back a chunk of memory that is not in use anymore to the list of available chunks. When the *free* is called to a specific pointer, this reads the metadata of the chunk (which are immediately before the address of the pointer) and puts it back in the bucket of free chunks, eventually coalescing it with another free chunk if these

[1] `http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/`

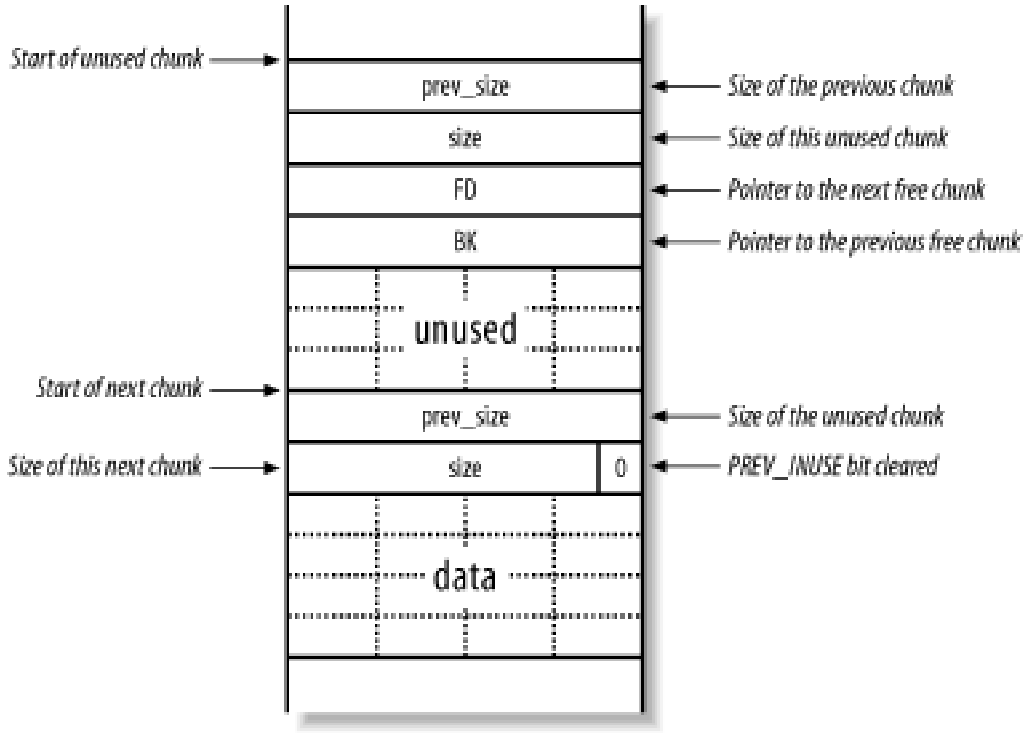


Figure 4. Scheme of the content of a used and a free chunk. Image taken from[1].

should be adjacent [44, p.91].
Both for *free* and for *malloc* there are many different implementations that deal with problems related to optimization and algorithms in a different way, but since these is outside the scope of this thesis it will not be discussed.

3.5 Attacks to binaries

After having discussed some background information about the architecture and some inner mechanisms of the systems in object the attacks that this project aims to prevent or detect will be analyzed. This section does not want to be exhaustive and its sole purpose is to define the threat landscape in which TED operates. Because of this, only the simplest and most relevant attacks are presented.

3.5.1 Stack Overflow

Stack overflows are one of the oldest software vulnerabilities, since they are tightly related with C/C++ code that is around since the early days of modern operating systems. Despite this, it is still one of the most common and exploited vulnerabilities [53] [91], with dozens of new discoveries every year. The most common form of this attack involves overflowing buffers that are on the stack. As the name describes quite effectively, the attack consists of stuffing more data into a buffer than the amount it can legitimately hold. It is relatively easy to make programming mistakes that lead to this kind of vulnerability, but the effects are not as trivial: a successful exploitation of a stack overflow can lead to denial of service or remote code execution, possibly compromising the entire machine [44].

How stack overflow works. The logic of stack overflow relies on how parameters and return address are saved into the stack. As discussed in section 3.4.1, when a function is called, its arguments are pushed into the stack if they belong to the MEMORY class, then the address of the next instruction to be executed is pushed and then, in the prologue of the called function, the value of the frame pointer is also saved on the stack. It has also been mentioned that the stack grows backwards and that it hosts local variables not allocated through *malloc()* or analogue mechanisms.
The attack consists then in exploiting local buffers that are saved on the stack and overflowing them, overwriting the whole buffer, the saved frame pointer and most important the return address. If an attacker has the possibility to write the return address, it means that he controls what the processor will execute. In the most sophisticated case, the attacker will redirect the execution to a portion of code that he wrote (for example into the legitimate content of the buffer). This code is often called payload, and consists in most of the cases of *shellcode*, which means code that spawns a shell for the attacker, granting him control over the machine [44, ch.3] [70].

Problems related to the stack overflow. In this discussion it will be assumed that the objective of the attacker is to get a shell from the victim by exploiting a stack overflow vulnerability. What the attacker has to do is to first produce a shellcode that fits comfortably into the buffer, then he has to supply this code and eventual padding to the vulnerable program and overwrite the return address with the address at which the supplied shellcode begins. From this it is possible to notice one challenging problem: guessing at what address the shellcode will be. In the (weak) assumption that the stack starts at the same address for every program, it would be possible for the attacker to get this address and then guess the one where the shellcode starts (the address of the buffer the attacker is overflowing) [44, p.27-33]. This process is tedious and not trivial, so there are some methods to facilitate it. The most common is the NOP method [44, p. 33-35] [70], which consists of filling half the buffer, before the shellcode, with NOP instructions. NOP is an instruction used to delay the execution doing nothing [20, ch. 2, p.48]; this is very helpful in exploiting a buffer overflow since the attacker does not have anymore to guess the exact address at which the payload

begins, but any address between the beginning of the NOP pad (also called NOP sled [44, p.33]) will lead to successful execution of the payload, making the choice of the address with which overwriting the return address memory location way more flexible.
As it will be discussed in section 3.6, the assumption of having a stack starting at the same address for every program is not only weak but obsolete, therefore it is worth mentioning, because it allows to bypass this exact problem, a technique called *ret2libc*. In this attack the attacker rather than providing a payload to be executed, will call a *libc* common function that will spawn a shell or perform other operations in his behalf. In this case all what the attacker does is providing arguments (on registers or on the stack) to the chosen *libc* function (*system()* is a common choice) and then call this function by overwriting the return address with its address. Even in this case it is clear that knowing the address of the chosen *libc* function (and also of *exit()* and other possible arguments, such as the string "/bin/sh" in case of a shell) represents a major challenge [44, p.35-38].

3.5.2 Heap Overflow

It is possible to refer to heap overflow whenever an attacker can corrupt memory that it is not on the stack. Often this attack is just referred to the one against a *malloc()* implementation, although its scope is much bigger than that. With a concept similar to stack overflow, the attack is based on the possibility to corrupt information which is stored right after the location where it is legitimately possible to write. As it has been described in section 3.4.2, the chunks of memory managed by *malloc()* and *free()* contain at least the size of the chunk (with some flags) when they are in use, and even more information when they are free. The idea behind the attack is to overflow one buffer allocated with *malloc()* and overwriting the metadata of another chunk. This will lead to *free()* or *malloc()* making illegal access to memory and will allow the attacker to write in arbitrary locations [44, p.94-96].

Logic of the heap overflow. Heap overflows exploitation is generally more complicated compared to stack overflows, although the logic is relatively similar, and in order to explain it the simplest example will be taken in consideration. The case in question exploits a macro present in *malloc.c* that is used for removing a chunk from the double-linked list of free chunks, but also to merge two adjacent free chunks. The macro is called *unlink* and what it does in practice is the following:

1. When a chunk is freed and needs to be merged *unlink()* is called.

2. Set the pointer to the next free chunk (FD) to *P+8, which means simply the actual address to the next free chunk.

3. Set the pointer to the previous free chunk (BK) to *P+12, which means simply the actual address to the previous free chunk.

4. Set the pointer at FD+12, which is the BK pointer for the next free chunk, to BK.

5. Set the pointer at BK+8, which is the FD pointer for the previous free chunk, to FD.

This macro can be exploited because it gives the possibility to write at FD+12 the content of BK [46]. In practice this is done tricking the *malloc()* algorithm in cases where two adjacent chunks are allocated, with the possibility to overflow the first. If this condition is met, an attacker can proceed in multiple ways. A standard way is to modify the chunk header of the second buffer with a negative value for the size of the previous chunk -so that *free()* will eventually consider as "previous chunk" the second buffer that the attacker controls-, a negative value for its own size and then controlling the FD and BK pointers. This trick in fact allows to write the content of BK into the address provided at FD+12 [44, p.95-97]. One of the possible ways to exploit this is to choose as BK an address for the stack (assuming that is executable) and for FD an address of a function like *exit()* and subtract to the latter 12. In this way when *exit()* or the overwritten function is executed, the control will go to the content of the memory pointed on the stack [44, p.97-98]. It is important to note that this specific exploit will not generally work on modern systems that have been hardened, specifically the *unlink* macro has been changed, and it is only functional to explain the logic behind heap overflows.

3.5.3 Architecture based attacks

In this section will be briefly discussed some attacks that are dependent on the architecture of a given machine, specifically Spectre and Meltdown attacks will be described, while other attacks, such as Rowhammer, will be left for future work. Spectre and Meltdown attacks have been just recently discovered at the beginning of 2018 and several variants have been presented. A common characteristic is to exploit hardware features such as speculative or out-of-order execution to read memory content. In 2017 researchers discovered three ways to determine valid kernel addresses (which are supposed to be randomized to provide security) by using double page-faults [51], software prefetch instructions [48] or Intel hardware transactional memory [54]. Following these discoveries, researches found out that it is possible not only to determine the addresses of the kernel but also to exploit such hardware features to read the content of the memory. In the case of Spectre attack [57] the branch prediction feature is exploited to read the victim process' memory content. In Meltdown [57] [61] attack, which is considered a variant of Spectre, the out-of-order execution feature is exploited allowing the attacker to read the entire memory of a process, of the kernel and the physical addresses mapped inside the kernel space. It is important to note that in addition to constitute serious security issues, these attacks can be exploited to break defense measures that rely on address randomization and consequently facilitate other kind of attacks such those described in the previous sections.

3.5.4 Reverse engineering

This section will describe briefly why reverse engineering can be considered a form of attack in some cases. Reverse engineering is a practice in which a man-made object is analyzed to gain knowledge on the design or to extract information [39]. Reverse engineering a binary means disassembling it, debugging it and/or attaching to the process associated with its execution, in order to understand its behavior or to recreate the source code. The reason why in some cases reverse engineering constitutes an attack is that it might be used to violate intellectual property, such as algorithms, or to facilitate piracy and unlicensed copies of software. In the context of this thesis, the main danger represented by reverse engineering, in addition to the ones mentioned, is when this is used to circumvent access restriction, in cases where this is provided in some form by a binary. It is clear then that despite reverse engineering is not an attack *per se*, it might be used for different purposes depending on the individual cases and therefore needs to be taken in consideration.

3.6 Defenses

In this section the main theoretical core of this study will be discussed. After having presented an overview of the structure of executable files, their memory layout and the most common and basic exploitation techniques, the defense mechanisms available for Linux systems against such attacks will be described.
The defense techniques in the context of this thesis are divided in three main groups:

1. System wide protections (from now on type *A*).
2. Executable specific protections against vulnerability exploit (from now on type *B*).
3. Executable specific protections against reverse engineering exploit (from now on type *C*).

The first group includes measures that are used by the kernel, that are applied to all the executables running in a given machine or that are applied as system settings.
The second group includes those measures that are applied individually to each executable, for example during or after compilation time, and that aim to protect such binary from exploits that target vulnerabilities in the code.
Finally, the last group includes those measures that are applied to each individual ELF file with the aim of mitigating exploits in which the attacker disassemble, debugs or attaches to the process associated with the execution of a binary with the purpose of gaining confidential information or to force the binary into a non standard behavior.
Before enumerating the different defenses taken in consideration in this research, it is necessary to premise that a large number of specific and custom tools exist that implement several protection mechanisms. For this reason, the list cannot be considered completely exhaustive.
The mechanisms taken in consideration are summarized in Table 4.

3.6.1 Libsafe

Libsafe is a dynamic library that implements many standard *libc* functions in a safe way [24].
This is made to avoid attacks such as buffer overflows that derive from the use of insecure functions without a proper boundary check. Some of the functions that are considered unsafe are: *strcpy*, *strcat*, *getwd*, *gets*, *fscanf*, *scanf*, *realpath*, and *sprintf* as pointed out in [24], in addition to *vsprintf*, *fscanf* and *sscanf* [41, p.275].
If Libsafe is present in the system, it intercepts all the calls made to unsafe libraries and executes the rewritten -safe- code instead. The safe implementation in Libsafe is able to detect at runtime what is the maximum size of the local buffer in which the program is going

Table 4. Defense mechanisms against various exploitation attacks.

Name	**Type**	**Protection against**
Libsafe	A	Stack overflow attacks
Stack Smashing Protector (SSP)	B	Stack overflows
AAAS	A	ret2libc attacks
Non-executable pages (NX)	A/B	Code Injection based attacks
(K)ASLR	A	Exploits that need advanced knowledge of addresses
Safe Unlink	A/B	Basic Heap overflow attacks
Stripping	C	Debugging - Reverse Engineering
KPTI	A	Meltdown attack

to write; this size is estimated assuming that the buffer cannot extend beyond the current stack frame, therefore can be considered an upper-bound of the actual size. Once this limit is established, every access to addresses past it will result in an error [24].
The protection offered by Libsafe is quite basic. In fact, it only protects against a certain type of stack overflow attacks, namely the ones that rely on overwriting the return address, as described in section 3.5.1. This type of attack is one of the simplest to perform and also to defend from, but many more are available that do not need to overwrite the return address to hijack the control [44, p.377-381]. Groups of researchers have extended Libsafe since its first appearance in 2001, and several versions of it exist. One of this is LibsafeXP [60] which enforces the same protection of Libsafe, but extends it also to global variables and dynamically allocated buffers on the heap. It is clear that LibsafeXP protects against a wider range of attacks compared to Libsafe, but once again the protection is limited to exploit misusing *glibc* functions and more important, the same logic to compute the expected maximum size of stack buffers is used, and therefore the same weaknesses are inherited.
Considering the available literature, it seems unlikely that Libsafe and derived tools are commonly deployed in actual systems.

3.6.2 Stack Smashing Protector (SSP)

The Stack Smashing Protector, originally developed as ProPolice [44, p.388] and later rewritten in a GCC-friendly way, is a protection mechanism that is integrated in compilers (such as GCC) and is based on the concept of canaries. Canaries are special values, that are placed on the stack and checked to detect if a stack overflow occurred. This means that SSP can be

used just to detect that an exploitation happened, not to prevent it. The value chosen for the canary depends on the implementation; various models have been proposed: NULL canaries, *Terminator* canaries (containing a mix of null byte, CR (Carriage Return), LF (Line Feed) and %
xff that are general string terminators for many C functions), random canaries and random XOR (Exclusive OR) canaries, where a random value is XOR'd with a control data [73]. In the case of SSP the random canary model is used [44, p.388-389] in the following way [44, p.390] [58]:

1. SSP rearranges local variables and function arguments, copying the latter.
2. All the buffers, which represent the main target for overflows, are placed at higher addresses (lower in the stack) whereas local variables and the copies of the function arguments are placed at lower addresses.
3. The canary is placed right after the buffers, at higher addresses.
4. Saved registers (for example %rbp and %rsp) are not relocated, so they sit after the buffers and they can be overwritten. However, the canary is checked before accessing them.

The result of the stack rearranging is what in the context of SSP is called *Ideal Stack Layout*. On top of the stack there are local variables and the used values for the function arguments, then there are all the buffers, then the canary, then the saved registers, the return address and finally the unused copy of the function arguments. With this setup if a buffer overflow occurs, the canary value will be overwritten and when this will be checked on function exit, the test will fail and the overflow will be detected consequently terminating the execution. Despite the fact that SSP offers a solid protection against many types of stack overflow attacks [73] there are still cases in which the SSP is partially or totally ineffective. Some of these cases are:

1. Multiple buffers. All the buffers are placed at the higher addresses, meaning that by overflowing one buffer it is possible to overwrite the content of another one. This situation might be dangerous depending on the logic of the program [44, p.391].
2. Buffers smaller than 8 elements. Overflows of such small buffers will not be detected [73].
3. *Structs* containing buffers. It is not possible to rearrange the member of a *struct*, meaning that if this contains a buffer, this can potentially be used to overflow all the variables defined after it in the *struct* [44, p.390] [73].

4. Functions with variable number of arguments. Since the number of arguments in this case is not known at compilation time, it is not possible to copy and relocate the arguments for such functions [44, p.390].

5. Buffers created with *alloca()*. In this case, the buffer is placed on top of the stack and *de facto* invalidates the protection on all the local variables [44, p.390].

6. Variables that can be overflown used within the function call. This case is slightly more complex than the previous ones and it depends from the fact that the canary value is checked on function exit. If one argument that can be overflown is used before the function terminates, for example to reference an object, the exploitation will be detected too late, after the attacker hijacked the control [44, p.392].

In addition to the cases listed above, which are somehow intrinsic in the design of SSP, attacks to overcome the protection offered by SSP have been developed. One case specifically seems to be relevant, and concerns mostly web servers that use *fork()* to handle each connection. In this specific case in fact, the children processes inherit the canary value belonging to the father process, rendering easy for an attacker to bruteforce it. This problem is well outlined by Marco-Gisbert et al. [64] who propose a new version of SSP called RAF SSP, Renew After Fork SSP. In their implementation a new canary value is assigned each time a process is spawned with *fork()*, and therefore it is not possible for an attacker to launch multiple connection probing all the space of canary values, since these will differ for each connection. Despite this improved version solves a major problem, the implementation consisted in rewriting the *fork()* function, and the RAF SSP can therefore be used just by preloading a shared library; at the best of the author's knowledge, RAF SSP did not manage to replace the standard SSP.
Despite the mentioned limitations, SSP protects effectively against a wide range of stack overflow types, as outlined in [73], specifically better than its precursors such as StackGuard [72] or StackShield [73].
It is worth mentioning that over the years many other solutions against stack overflow-based attacks have been presented and designed. One common strategy used by both Younan et al. [90] and Solanki et al. [78] includes the use of *Shadow Stacks*. A shadow stack is an artificial stack, or an array of them, that is used to ensure and verify the integrity of the original stack. Younan et al. [90] implemented a mechanism in which the data types are divided in nine categories based on the likelihood of being used as an attack vector or as a target, and based on this they are assigned in one of the corresponding nine stacks. These are separated by a page guard to avoid the traversing from one stack to another. In this way, an overflow in a vulnerable buffer cannot compromise the return address or any other sensible

target. In order to use this mechanism, Younan et al. provided a C compiler.
The same concept of shadow stack is also employed in the more recent Secure Patrol [78]. In this case there are two shadow stacks which are used to memorize the return address and the content of the base pointer, respectively. When a function returns, these values are compared with the ones on the actual stack. If a difference is detected, the program terminates with an error. In addition, the Global Offset Table is protected in a similar way, by using a global array that mirrors the its content. Every time the GOT is consulted, the entry is compared with the one in the mirrored version and if these differs, the program is terminated with an error. Once again, none of these mechanisms seem to be widely deployed, probably for the heavy constraint imposed by using custom compilers and/or kernels.

3.6.3 Non executable pages (NX)

Many basic versions of exploits inject a payload that the attacker wants to be executed, such as a shellcode, as discussed in section 3.5.1. These attacks are possible because the attacker is able to execute code that resides on the stack; therefore one of the strongest security measures is making the stack non executable. In fact, it is possible to go even further and mark as executable only the pages of those sections that contain code. In this case, sections such as stack, heap, .bss and others are marked as non executable, reducing quite effectively the possibility to perform foreign code injection attacks [44, p.383]. Applying the same logic, NX has been extended with a principle that is commonly written as *W^X*, which stands for Writable XOR Executable. W^X refers to the fact that pages which are writable should not be executable and vice versa, pages that are executable should not be writable (e.g. it should not be possible to write on .text section) [44, p.381].

PaX. Both NX and W^X are implemented (not exclusively) in a kernel patch called PaX [81] [83]. The main purpose of PaX is to put together a set of defensive mechanisms in order to limit the possibility for an attacker to [83]: inject and/or execute arbitrary code, execute legit code with a flow different from the original or execute legit code with the correct flow but with arbitrary data. The NX represents a good measure in the first scenario, and therefore PaX implements it. Specifically, the feature of PaX that enforces NX is called NOEXEC, which is split in two groups of features: PAGEEXEC + SEGMEXEC and MPROTECT.
PAGEEXEC and SEGMEXEC are used to implement the actual non-executable page logic, since in IA-32 architectures the NX bit was not natively supported. Because in 64bit architectures the NX bit is natively supported, PAGEEXEC and SEGMEXEC are not relevant in this discussion. MPROTECT on the other hand is used to enforce W^X and to forbid the introduction of new executable code into the process address space. In order to do so, *mmap()*

and *mprotect()* interfaces are restricted so that the following actions are not allowed [80]:

1. Create anonymous executable mappings.
2. Create W+X mappings (both writable and executable).
3. Making an executable or read-only mapping writable, with the exception of relocation on a shared library.
4. Making a writable or readable mapping executable.

It is not easy to measure how commonly PaX is applied in servers deployment, but several other implementations of the NX and WˆX principles are commonly used in default Linux installations.

3.6.4 Address Space Layour Randomization (ASLR)

ASLR is probably the most important security measure to mitigate memory corruption attacks. The main idea behind ASLR (and of the similar KASLR discussed in section 3.6.8) is to introduce randomness into the addresses assigned to a given process, including its stack, heap and library addresses [44, p.396] [82]. ASLR was first introduced by PaX team in 2001 therefore the PaX implementation will be discussed, despite the fact that multiple mechanisms that randomize addresses exist.
The randomized memory layout is determined when the process is created, by generating three random seeds that are then used to apply different randomness to addresses belonging to three main categories: *delta_exec*, *delta_mmap* and *delta_stack*. The randomization then is performed as follows [82]:

1. Main code, data, .bss segments and the *brk()* managed heap are randomized using *delta_exec*. The feature that takes care of this is RANDEXEC or RANDMMAP.
2. The *mmap()* managed memory, including libraries, heap, thread stacks and shared memory is randomized using *delta_mmap*. The feature that takes care of this is RANDMMAP.
3. User and kernel stacks are randomized using *delta_stack*. The features that perform this task are RANDUSTACK and RANDKSTACK, respectively.

In each case the seed computed is used in a different fashion to produce the desired result.
A very important element to consider about ASLR is the difficulty for an attacker to guess a given address. This strictly depends from the number of random bits in one address, which is

a constant defined per each of the above-mentioned categories [82]. If the number of random bits is not large enough, a given address could be brute forced. In particular, Shacham et al. [75] showed how, in 32-bit architectures, it was possible to break PaX ASLR protection in only 216 seconds on average. The method used is very similar to the one used to defeat random canaries: children process spawned with *fork()* inherit the random seed and therefore allow the attacker to brute force addresses. The root of the above weakness of ASLR, as it has been pointed out, is the small number of bits of entropy in the addresses used. This problem existed in 32-bit architectures but appears to be solved in 64-bit ones where, thanks to the larger number of bits available, for a given address there are at least 32 bits of entropy. In the context of this project, it is assumed to be running in a 64bit environment and therefore also that it is not possible to brute force addresses in order to perform attacks like return-to-libc. However, this does not solve all the problems related to ASLR. Wang et al. [88] showed another derandomization attack against ASLR that did not involve brute forcing address but only the concept of ROP (Return Oriented Programming), which uses small parts of legit code (gadgets) followed by *RET* instruction and combines them together to perform arbitrary computations. A more advanced attack has been showed by Snow et al. in 2013 [77], called Just-In-Time Code Reuse (JIT Code Reuse) in which the attacker exploits a memory disclosure to map the program's address space at run time and to find necessary gadgets. Several other attacks have been designed to exploit the powerful ROP, and in order to contrast them, researches have proposed either improvements to ASLR or brand new randomization schemes [23] [32] [35] [49] [62] [71]. One of the most recent proposals has been developed by Chen et al. [30], called JIT-ASR. In this proposal the randomization is achieved by altering the Virtual Page Number of the virtual addresses, while leaving the physical address unmodified. This alteration is done continuously at every access with the idea that even if the attacker should obtain or guess a memory address, this will have become stale and unused at the next access.
Once again, despite the very promising results both from the defense perspective and from the performance impact, it seems that none of the mentioned solution is currently integrated in the common operating systems and replaced the default and relatively basic version of ASLR.

3.6.5 ASCII Armored Address Space (AAAS)

AAAS is a basic protection against attacks such as ret2libc, or more in general attacks in which a system library call is misused while exploiting a bug in the code. The way AAAS works is very simple: some important system libraries functions are mapped to addresses that contain NUL values [44, p.394]. Specifically, *Glibc* maps the *libc* shared library functions to

what is called "*armor zone*", meaning the first 16MB of memory, whose addresses contain NUL bytes at the beginning [27]. The reason behind AAAS is that functions like *strcpy()* use the NUL byte as string terminator. This means that if it is possible to overwrite memory because of a misused *strcpy()* function, if the address of a system library is used (like *execve()*), the attacker will be able to copy data only until the end of this address, when the NUL byte will be reached [44, p.394].
AAAS is not a very strong security measure as there are several ways to bypass it, especially in small-endian systems, and in general the main executable is not moved to the armor zone, leaving a lot of code "unprotected" [44, p.395]. For this reason, systems should rely on ASLR instead.

3.6.6 Safe Unlink Macro and heap protection

The safe *unlink* macro is more properly a code fix than a security measure. The *unlink* macro has been first mentioned in section 3.5.2 as one of the many ways to perform heap overflows attacks, and its fix is one of the very few common measures to mitigate them. All modern systems are expected to use the safe version included in *Glibc* since 2004, which performs an integrity check on the double-linked bin list, before unlinking the block it is working with [18]. This simple improvement should be taken for granted on a modern system and effectively stops the heap overflow flavor described in section 3.5.2. However, it is important to remember that many more ways to perform heap overflows exist and keep being exploited, and despite this fact, there are not many defenses that can be applied to protect a system from them.
In fact, despite the number of heap-based vulnerability discovered every year is growing [11], there is not a standard protection tool similar to, for instance, Stack Smashing Protector that takes care of enforcing at least a basic heap protection. Over the years, several techniques have been designed and proposed mainly falling in two categories: canary based protection and secure allocators. To the first category belong tools which use the same concept of ProPolice: special values are placed before and after the memory chunks that can be overflown and are later verified, aborting the execution in case those canaries are found modified. This is the case for example of Van Acker et al. [85] and more recently of Zeng et al. [92] with the difference that in the latter case there is a thread that "cruises" over the user address space to verify the integrity of the canaries asynchronously. More recently canaries have been used also by Nikiforakis et al. [67], but in this proposal their integrity is verified with the assistance of the kernel every time a system call is requested by the process.

The other approach, as mentioned before, is to provide a secure allocator. This is the case

of DieHarder [68], based on DieHard [25], which provides a probabilistic memory safety, which means taking advantage of the vast virtual address space to place allocated blocks of memory far from each other, providing therefore a very small chance that two blocks will be adjacent and therefore that it will be possible to overflow one of them from the other. In addition to this, DieHarder provides features such as filling free'd objects with random data. Instead of a fully new allocator, Younan et al. [89] proposed a modification of *dlmalloc* in which control metadata and data are stored separately. Finally, Hsu et al. [50] proposed Heap Spray Protector, a tool meant to counter one specific heap-based attack, the heap spray. In this attack, the attacker writes hundreds of times the shellcode he wants to execute in order to make more likely to guess the address to which transfer the control. The way Heap Spray Protector enforces the defense is to limit the instruction (*int 80* for older Linux versions) used to perform a system call. With this constraint it is useless for the attacker to inject shellcodes multiple times (which contain at least a system call). Their implementation is done as part of *Glibc* and the kernel.
As it has been anticipated before, none of these technique is widely deployed and can be expected to be present on a given server. The reasons are probably multi-faced, from performances to compatibility with other code. In many cases administrators might not be willing to apply a custom version of *Glibc* or to have a custom kernel module to avoid maintenance issues or incompatibility with future versions. In addition, all the defense measures that are not applied on top of the allocators, but are integrated in one, such as [68] and [89], need to be considered on a per executable basis. What this means is that every developer has the freedom to develop or use the allocator it suits him more, without the need to use the system's default one. In the context of this project, it is meaningless to consider a system safe from heap-based attacks just because the default system allocator is secure. Every executable can use its own -potentially unsafe- allocator, and therefore this is the condition that should be verified.

3.6.7 Stripping binary

Stripping a binary is an action that belongs to a broader category of measures used to protect binaries. These measures aim to offer protection rather than from memory exploitation attacks, such as buffer overflows, from reverse engineering.
In order to provide protection from reverse engineering, several approaches exist, summarized very efficiently by Kim et al. [56].

Anti-Debugging. In this category fall all those measures used to prevent the use of or to confuse debuggers.

Packing. Packing a binary usually means compressing the content of the executable and adding a stub procedure that takes care at runtime to decompress and execute the file [40].

Encryption. The use of encryption is similar to packing, with the difference that the code is not compressed but encrypted, meaning a better protection at the price of a bigger overhead [56].

Self-checking. A program performs self-checks when it verifies the integrity of its own code, making sure no instructions have been added or removed.

Tool-detection. A program can perform tool detection by extracting some system information with the aim of understanding whether some specific tool (ex. a debugger) is also running.

Obfuscation. Obfuscating the code of a program is the most popular technique used to protect a binary, and it involves generating confusing and useless code with the aim of making the understanding of the actual code harder.

Binary modification. Modifying the binary is exactly what stripping the binary does. In this case debugging and all information not relevant for the execution are removed from the binary.

The area in which more research has been made is obfuscation. There are, in fact, many researches which study how to generate code which is very hard to disassemble. Blazy et al. [26] proposed recently a generic method that can be used to obfuscate the data of a program, but much more work has been made during the years [29] [33]. Unfortunately, software obfuscation is not really interesting in the context of this project. The reason for this is that TED works with binaries and is agnostic about the logic of the code itself. This means that given am ELF file, it is not possible to determine whether this is the compiled version of an obfuscated program. The literature, at the best of the author's knowledge, does not provide any technique or tool to detect whether a program has been obfuscated in an automated way and therefore the tool proposed here will not attempt to perform such operation. It might still be possible to guess whether a program has been obfuscated by a manual inspection, but in general this is out of the scope of this thesis.
Besides obfuscation, not much work has been done in the other areas mentioned. One possible reason for this is that countering reverse engineering is usually a target for malware

authors and for software houses producing proprietary software. In both of these cases, the Linux market is relatively small, and therefore the effort cannot be compared to the one put in protecting Windows executables, for example. In this regard, one interesting observation is that as of today, there are several solutions to protect Windows binaries, but not a standard tool to protect ELF64 files. UPX[1] is one of the few tools that can be considered, but its aim is not to protect the binary but just to pack it, mainly to save disk space. This might be useful in resources-constrained environments such as embedded devices, but it is virtually of no use for binaries running on servers. The problem of the lack of a secure binary protector has been already pointed out in 2010 by Kim et al. [56] who in fact proposed two tools: an improved version of UPX (called secure UPX) and AES-encryption packing. The context of their research was embedded software and therefore the goals were different, however, both the tools are not publicly available and therefore it is reasonable to think they are not used either.

Among the other tools present in the literature there are Burneye [2], released in 2001 and not maintained since 2002, that was aimed at protecting x86 ELF binaries, and Shiva [76], a similar tool also designed for x86 ELFs that was released in 2003 and whose protection apparently has been broken [38]. Finally, one tool has been described in [40], despite it cannot be found anywhere: Maya's Veil. This tool allegedly implements advanced functions and includes protection for x64 ELFs. Unfortunately, given its unavailability, there is no reason to suppose it is being used.

It is possible that the lack of Linux binary protectors, besides the small number of closed-source software and the lower amount of malwares from which new strategies can be learned, depends on the fact that a complete protection against reverse engineering cannot be achieved. What it has been achieved so far is to slow the reverse engineering process, and also some techniques such as anti-debugging rely on specific software bugs or characteristics, rather than on general principles [45]. All these elements combined might contribute to make the design and implementation of ELF binary protector not appealing or not worth the efforts in an environment that runs at a very fast pace such as Information Security.

Whatever are the reasons, in this research it is possible to consider stripping a binary the sole measure to protect a binary from reverse engineering. It is clear that this is a very basic measure but it is also the only one that can be reasonably assumed to be commonly used. A stripped binary in general will not contain any symbol information [16] [40, p.29-31] thus making reverse engineering harder because of the lack of names for functions and variables. It is important to stress that stripping is not mainly meant as a protection mechanism and its real aim is once again to save disk space.

[1]`https://upx.github.io/`

3.6.8 KASLR and KPTI/KAISER

Similarly to what ASLR does, the kernel address space is also randomized at boot time. The feature that performs such operation is called KASLR, which stands for Kernel Address Space Layout Randomization. This feature is meant to protect from the attacks discussed in section 3.5 targeting the kernel. Therefore KASLR provides a protection to the kernel similar to the one ASLR provides to user-space binaries. However, with the new discoveries mentioned in section 3.5.3, the security provided by KASLR has been declared insufficient.
To solve this problem, KAISER has been designed in 2017 [47]. KAISER stands for Kernel Address Isolation to have Side-channels Efficiently Removed and aims to counter the side-channels discovered to map Kernel addresses using what has been called a *shadow address space*. The practical implementation of KAISER is called Kernel Page Table Isolation (KPTI). In a KPTI protected system, every process has two address spaces. In one, the user space and only a small necessary (for signal handling, for example) portion of kernel space is mapped, while the kernel space is not mapped. In the second, the kernel space is mapped and the user space is mapped but protected with SMAP and SMEP (Supervisor mode access prevention and Supervisor mode execution prevention) which do not allow to read or execute code from memory belonging to user-space while running in kernel space.
It has been shown that KAISER can effectively stop the side-channels attacks reported in [48] [51] and [54], and it also incidentally mitigates Meltdown attack [61], as reported by Lipp et al. who stressed the need for the systems to include KAISER/KPTI as soon as possible. It is important to remember that KAISER does not provide protection against other variants of Spectre, for which there are no yet effective security measures and the ones developed, mostly consisting in kernel patches, have a significant performance overhead.

3.7 Containers

In this section a brief overview of what containers are and their main features is presented, with the aim to highlight why their use in this project is beneficial.
Containers in a general meaning are encapsulation of an application with its dependencies. One main feature of containers is that they share resources with the host, leading to a more efficient resource handling compared to virtual machines. In fact, containers do not use any supervisor to virtualize the hardware, as shown in Figure 5, and instead they share the kernel with the host. What this means is that containers are more lightweight than full virtual machines, and it is possible to run dozens of them on a single host, but at the same time this means also that containers are forced to run the same kernel of the host [66, p.2-4].

[1] `http://content.serverspace.co.uk/hubfs/Blog_Images/container-vs-vm.jpg`

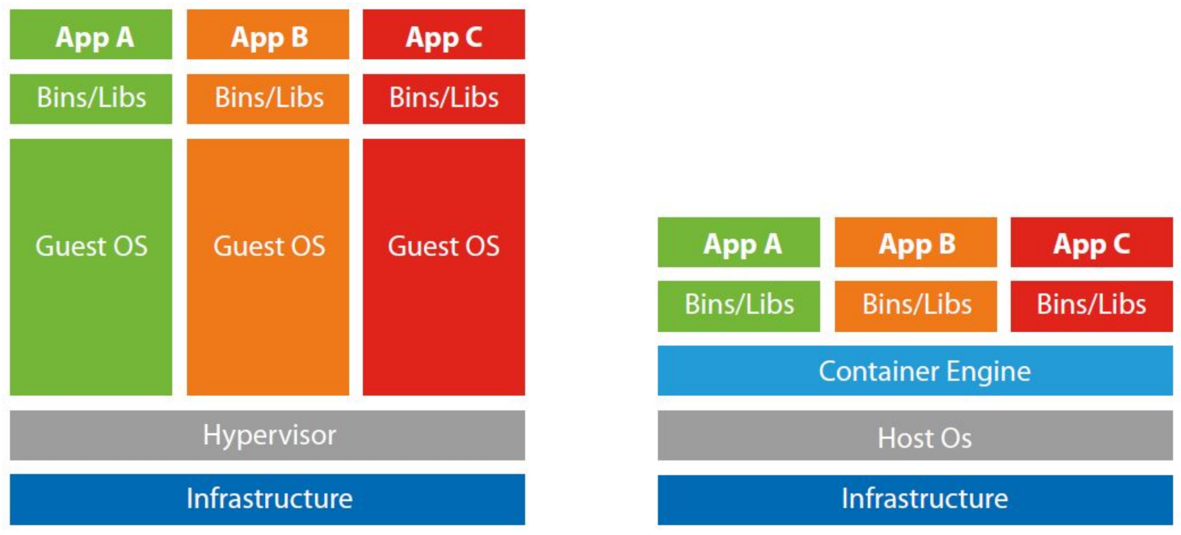


Figure 5. Comparison scheme between Virtual machines and containers virtualisation. Image taken from[1].

To achieve the containerization of processes and the encapsulation of application, all the container engines rely mainly on two kernel features: Cgroups and namespaces.

Cgroups. Cgroups [3] are used to manage the resources (CPU, memory, etc.) allocated for containers, including CPU, memory, network and so on.

Namespaces. Namespaces are used as a mechanism to ensure that hostname, users, networking and processes of different containers are isolated. The UID of users is the same on the host and inside a container, while the names are separated, leading to the possibility for the same user to have different UIDs. The use of namespaces allows, from Linux 3.8, to unprivileged users to create nested namespaces in which they have privileges [21]. To be more specific, in Linux six different namespaces are implemented:

1. Mount namespaces isolate the set of filesystem mount points seen by a group of processes. This means that processes in different mount namespaces might see the filesystem in a different way. With the introduction of this namespace, *mount()* and *umount()* also started to operate in the namespace associated with the process calling them [21].

2. UTS (UNIX Time Sharing) namespaces isolate hostname and domain name returned by the *uname()* system call. Each container can have its own hostname and NIS (Network Information Service) domain name [21].

3. IPC (Interprocess Communications) namespaces isolate inter-process communication.

4. PID (Process ID) namespaces isolate the process ID. Each container specifically can run its own *init* process (PID 1). From the perspective of a specific PID namespace, a process has two IDs, the one inside the namespace, and the one in the host. The PID can therefore be nested and it is possible to send signals to other processes with *kill()* if these are inside the same namespace or in a namespace nested in a lower level [21].

5. Network namespaces isolate all the networking resources, including IP addresses, routing tables, */proc/net* directory, port numbers etc. [21].

6. User namespaces isolate UID (User ID) and GID (Group ID) numbers. This means that a user running a specific process can be different inside or outside a user namespace. As stated before, it is possible to have an unprivileged user outside, and a UID 0 inside the namespace, allowing a process to run privileged inside the user namespace, despite being unprivileged outside [21].

4 Methodology

This section defines the overall research process used to address the research questions presented in section 1.1. This process includes the choice of container technology to use, the selection of the defense mechanisms to test, the implementation of the tool, the verification and validation of the results obtained and the description of the conclusions drawn from the project.

4.1 Container engine selection

As a first step, the selection of the most appropriate container engine through a qualitative analysis is performed. In this analysis, the most popular platforms for running containers are examined taking in considerations several parameters and defining a related scoring system.

4.1.1 Container engine scoring system

The three aspects evaluated in a container engine are availability, functionality and performance, and to each container engine is assigned a score from 1 to 5 (in order to allow enough differentiation) respectively for each parameter. The availability includes evaluating what needs to be done to get and install the platform, what dependencies need to be satisfied and also whether there are public repositories from which container images can be obtained. The functionality consists of whether and how the container engine provides capabilities such as building images, collecting logs and events and sharing portion of the filesystem with the host. Finally, the performance evaluation includes the execution of a script that mocks one of the required use cases for the containers in this project: the directory */usr/bin* of the host is mounted inside the container. For each binary the content is copied on another file and the SHA256 hash of the file is computed. This process is executed measuring the execution time with Python *timeit* and is repeated 1000 times in order to get an accurate average. The average value is then compared with the native execution of the operation on the machine. If the time is lower than the native execution, the full score is assigned; if the execution time is longer than the native execution, 1 point is subtracted plus 1 additional point for each 10% of overhead.
Each one of these parameters will be weighted as explained in Table 5.
The sum of the weights is 10, and the weights are distributed in a sensible way according to the importance that the specific characteristic has in the context of this research.
The functionality is considered the most important criteria because a missing functionality in the engine will likely cause a lack of capability for the tool or an increase of its complexity.

Table 5. Weighted scoring system for Availability, Functionality and Performance for the evaluation of container engines solutions.

Parameter	Weight
Availability	3
Functionality	4.5
Performance	2.5

Moreover, limited functionalities impose a constraint on the future extension and development of this project. Because of this, a weight of 4.5 is assigned, making the functionality account for almost half of the global score.
Availability and performance are considered almost equals although the availability is estimated slightly more important for several reasons. First of all, the time required to fulfill the installation requirements might overweight the performance gains; in addition, a lack of availability in container images might make the project more complex or add some limitations (such as the need to host a private repository). Finally, since all the technologies in consideration are similar, the differences in terms of performances between the different technologies are expected to be minimal. For all the reasons above, a weight of 3.5 for availability and 2.5 for the performance reflects fairly the impact of each of them on the project.
The final score for each platform is computed with the following formula where A stands for Availability, F for functionality and P for Performance:

$$Final\_score = Score_A * Weight_A + Score_F * Weight_F + Score_P * Weight_P$$

Given the sum of the weights (10) and the maximum score for each criterion (5), the maximum value for the final score is 50.

4.2 Defense techniques selection.

The second step in the research process consists in choosing which checks, and the corresponding tools, to use for the vulnerability identification. This process is performed using a qualitative approach and making heavy use of the information collected in section 3.6. Given the very reduced number of defense mechanisms commonly deployed on real-life servers, it has been decided to limit the checks to the standard defense techniques with the idea of making TED usable in the real world. The vulnerability identification is therefore performed by verifying that the machine in object is making use of ASLR and that the binaries contained in it are stripped, compiled using canaries and respecting the principle of NX/W^X. In addition to this, considering the tight link between binary security and the system in which these binaries are executed, further tests are performed on a system level. The purpose of these checks is to verify whether the kernel running has some known vulnerabilities and if the system is vulnerable to the hardware dependent attacks mentioned in section 3.5.3. A more in depth description of the motivations for each of the choice mentioned can be found in section 5.3.

In order to facilitate the risk assessment, a scoring system is defined. To the result of each of the tests performed a score is assigned while two aggregated scores are defined for the system and for a single ELF respectively. The score has to be intended as a "vulnerability" score, meaning that a higher score implies a worse level of security. The reason for this approach is that it is easier to measure the risk and actual security issues found than measuring the security level negatively by the apparent lack of issues.

4.2.1 System score

The score of the system is computed on a total of 100 points, where 0 means that no security issues have been found and 100 means that the server has serious security issues. The choice of limiting the system score to 100 and to express therefore the score as a fraction has the purpose of giving an immediate idea of the security of the system. The score is then composed as follows.

Kernel exploits. For each confirmed kernel exploit found, a score of $15/2^{n-1}$ is assigned, whit $n \in [1, \infty]$, which represents the *n-th* exploit found. The first exploit for example will yield 15 points, the second 7.5, the third 3.25 and so on. The logic for potential exploits is the same, with the only difference that each one of them will contribute for $5/2^{n-1}$ points. The score assigned to confirmed exploits is limited to 30, while the one assigned to potential exploit is limited to 10. All together, the kernel exploits can yield a maximum of 40 points. The decision to set a limit for this score is motivated by the fact that often one vulnerability can lead to several exploits and therefore an unlimited increment would lead to an unrealistic high score, that does not reflect the real situation. The decremental scores are assigned for a similar reason; if the system is vulnerable to one exploit, the system is at severe risk. On the other hand, if the kernel is vulnerable to more than one exploit, the attack surface is more extended, but the risk does not increase linearly and every new exploit will aggravate slightly the already severe situation of the machine. Overall the kernel exploits contribute for 40% of the system score, which reflect the critical risk represented by a vulnerability in the kernel.

Hardware based exploits. Architecture dependent vulnerabilities often cannot be mitigated on operating system or software level; however, they do represent a threat for the system. For this reason, hardware based vulnerabilities can yield to a maximum of 20 points. As it will be described in section 5.3, three specifics architecture attacks will be tested: two variants of Spectre and a third variant also called Meltdown. Each variant of Spectre to which the system is vulnerable adds 5 points, while 10 points are added if the system is vulnerable to Meltdown. The uneven distribution for the points depends on the fact that Meltdown attack has a more severe potential impact. The important factor for which only 20 points are assigned to vulnerabilities that can potentially compromise the system is that as of today, there are no confirmed working exploits for such attacks and therefore the threat represented by such attacks is sever but potential.

ASLR. The presence and extent of ASLR in the system can lead to a maximum of 20 points, depending on how ASLR is enabled. The ASLR score is assigned as reported below:

1. 20 points if ASLR is disabled
2. 15 points if ASLR is partially enabled, only on stack, virtual dynamic shared object page, and shared memory.
3. 5 points if ASLR is enabled also for the heap and data segment.
4. 0 points if ASLR is enabled fully and additional patches have been applied.

NX support. If the CPU does not support the NX bit, then it might not be possible to comply with NX/W^X for single binaries, possibly voiding other ELF specific protections. For this reason, the lack of support from the CPU for the NX bit on memory pages adds 20 points to the system score. If the CPU supports the NX bit, no points are added.

4.2.2 ELF score

The score of each binary is assigned out of 100 points, where 0 means that all the security measures supported by TED are in place, while 100 means that none of them is and therefore the binary represents a potential vulnerability in the system. The details on the composition of the score are reported below.

NX. The prevention of execution for sections where there is no code is a very efficient security measure that mitigates a broad number of attacks. For this reason 50 points are assigned if the binary is compiled without complying with NX. Specifically, 30 points are assigned if the stack is marked as executable and 20 points are assigned if there is at least one section both writable and executable, contradicting W^X.

Canaries. Canaries represent another simple but effective security measure that counters a number of attacks. Because of this, the lack of canaries in a binary will add 40 points to the risk score for that binary. Specifically, 20 points will be assigned if no canaries are detected, and 20 points if the binary has not been compiled with the Stack Smashing Protector feature.

Stripping. The score assigned if a binary is not stripped is 10 points. The reason for such a low score is that stripping is a very basic measure to counter reverse engineering (specifically, it makes harder the debug of a program for the lack of symbols) and its main purpose is instead to save disk space. Moreover, stripping is not necessary in many cases, for example for all those binaries whose code is publicly available. In general, the severity of this score needs to be evaluated on a case by case basis, because for a specific binary it might represent an actual issue, while in the rest of the cases it might represent a mere commodity.

4.3 Implementation

After having designed the tool from a high-level perspective, the actual implementation of TED is performed, and a tool that respects the criteria defined in the previous sections is built.

4.4 Verification and validation of the results

In order to verify the validity and quality of the results obtained, the third step consists of practical tests and verification of what has been done. Two main types of tests are performed: a qualitative functional test and a quantitative benchmark. Finally, to further prove the usefulness of TED in the real world, a practical test on a real server and on a complex infrastructure will be executed.

4.4.1 Functional test

The first test has the aim of verifying that the tool produces the expected results. To verify this, a minimum of three different machines and operating system are tested using TED, and the reports are collected. After the operating system installation, for each machine, the following testing process is followed:

1. TED's dependencies are installed on the system.

2. TED is run on the system, the output report is collected.

3. The machines are tampered in a semi-random way. The purpose of this step is to introduce variable vulnerabilities in the system.

4. TED is run again on the systems.

5. The TED reports from the first execution are compared to the reports from the second execution.

The expected result of this test is that the difference between the two reports contains the changes made during the tampering of the machine. If these changes are detected, then it is reasonable to assume that TED performed according to its specifics.

4.4.2 Performance benchmark

The second test is a quantitative performance analysis with the purpose of measuring the overhead and performance loss that derives from the use of containers. This benchmark is useful to determine clearly what is the trade-off and if it is feasible to run the checks inside containers rather than natively. The benchmark consists of running and measuring the main checks both inside a container and on the native machine and compare the results. Specifically, the ASLR check, the kernel exploit test, the architecture exploit test and the ELF check are used. The ELF test is performed on a defined set of binaries and the average execution time is computed, while all the other tests consist in repeating the check for 100 times, in order to obtain an accurate average execution time. Once all the tests are done, the average value for the execution of each test inside a container is compared with the respective execution time on the native machine and the overhead is measured.

4.4.3 Real World test

To further test the possibility for TED to be used in real world situations, two practical tests are executed. The first consists in running TED on a real public server, while the second test involves running TED on a portion of the environment used for the Locked Shields 2018 NATO cyber exercise. In the case of real world environments it is not appropriate to perform a manual tampering of the server and verify that TED catches the changes made. However, this makes the verification of the results harder to prove for the lack of reference to compare TED's results to.

Public server. For what concerns the test on the public server, the verification of TED's effectiveness is based on a thorough inspection of the report and the related issues found. Depending on the findings, the machine might be hardened based on the information gathered and a second TED scan is performed once the hardening is complete. TED can be considered successful if it allowed to individuate critical issues and to establish an action plan to improve the security of the system, which is the standard proactive use case for this project.

Locked Shields infrastructure. Regarding the test on a portion of the infrastructure used for the Locked Shields 2018 cyber exercise, TED is used on the authorized machines with two objectives in mind. The first, in a similar fashion to the previous test, is to establish a vulnerability assessment of the machine and to catch possible security issues with a proactive approach. The second is to investigate possible rogue binaries or other related vulnerabilities that have been planted and/or hidden in the systems. The infrastructure on which the test is executed consists of two pairs of machines belonging to two different teams. The first two machines are undefended, meaning that no blue team performed any action on them, and therefore represent the reference to use for the comparison. The last two machines belong to the blue team who won the competition, and therefore it is expected that these machines had been hardened and they might contain different binaries. The result of the test is determined based on the comparison of TED's findings, comparing each undefended machine with the corresponding machine belonging to the winning team.

4.5 Conclusion

To conclude the research, all the results produced, with particular attention to the outcome of the testing and validation process, are observed and the conclusions on the project are drawn.

5 Implementation

The implementation of the tool can be divided in few logical phases, each of which will be described separately. The first phase, according to the methodology established, is the choice of the execution environment for the tool, meaning the container engine and also the programming language to use. The second phase is the choice of the defense mechanisms to check and the tools to use to accomplish this. The final phase is the actual code implementation.

5.1 Container engine selection

Despite being an old invention, containers are growing and developing only recently and therefore the market is still very young, leading to the flourishing of many container engines and orchestration systems. According to the methodology adopted, the evaluation criteria for container platforms are availability, functionality and performance.
In order to select the candidates that will be evaluated, a first preliminary selection of the available platforms needs to be performed. This selection will be performed with few criteria in mind: availability of the platform, availability and flexibility of images and maturity of the technology. The summary of the preliminary evaluation is shown in Table 6.
Two platforms have been excluded and will not be taken into account in the actual evaluation. OpenVZ is a platform similar to LXC for some aspects, but it has limited functionalities if it is not run on a specific kernel. This thesis aims to provide a tool which is agnostic of the system on which is run, and therefore OpenVZ would impose a serious constraint. Garden is a platform that aims to represent an alternative to Docker, but it is still in a early stage of development and it is virtually not used outside the Cloud Foundry environment.

Table 6. Summary of the container platforms evaluated to select candidates for this project.

Name	**Avail. platform**	**Avail. images**	**Maturity**	**Selected**
Docker	Pre-installed on some systems. Easy to install.	Public and private repositories.	Production ready and widely used.	X
LXC/LXD	Easy to install on most Linux distros.	Private repositories only for templates.	Production ready.	X
OpenVZ	Works best on its own kernel	Public templates	Production ready.	
Rkt	Easy to install on most distros. Preinstalled in CoreOS.	Can use Docker images	Production ready.	X
Garden	Various dependencies. Not stable installer.	Can run Docker images	Not used outside Cloud Foundry.	

The platforms that will be taken into consideration for this thesis and that will be evaluated in more detail are Docker, LXC/LXD and Rkt.

5.1.1 Decision-making process

From a practical perspective, in order to compare the container engines mentioned in section 5.1, the main characteristics of each of them will be outlined, with specific reference to availability and functionalities, whereas a benchmark will be performed to measure the performances, in line with the methodology presented in section 4.1.
The result of the evaluation is summarized in Table 10 on page 61 along with its outcome.
The benchmark will be performed on a Virtual Machine running a vanilla Debian 9 server distribution. The choice of Debian was made because it has one of the biggest market share among Linux distribution [86] and it is the base for the most used Linux distribution, Ubuntu, and therefore represents a common and representative scenario. For reference, one single execution of the benchmark script on the virtual machine takes on average 3.8s. The script -as it has been described in the methodology- executes a copy and computes the SHA256 hash for all the binaries inside the */usr/bin* folder.

5.1.2 Docker

Docker is the most popular container engines and the current industry standard. The summary of the scores assigned to Docker can be found in Table 7 below.

Table 7. Summary of the scores assigned to Docker container engine.

Parameter	**Score**
Availability	4
Functionality	5
Performance	5

The Docker architecture is based on few elements:

1. Docker daemon. The Docker daemon is responsible for creating, running, and monitoring containers and also for building and storing images. This daemon uses an "execution driver" to create containers. By default this is Docker's own *Runc*, but there is also the LXC legacy option.

2. Docker client. The Docker client is used to communicate to the daemon via HTTP, over a Unix socket or over a TCP (Transmission Control Protocol) socket. A rich API (Application Program Interface) is available to interact with the daemon.

3. Docker registries. Registries are used to store and distribute images. The Docker Hub is the default registry although it is possible to also run a private registry [66, p.35-36].

Availability. Docker can be installed in most cases from the package manager (*apt, yum, dnf* etc.) without any specific requirement. It does not need any specific Kernel feature but root access is necessary, since the Unix socket on which Docker binds itself by default is owned by root [5].
It is important to note that Docker is included in many cloud-oriented operating systems, such as coreOS Container Linux [4] and RancherOS [12]. The Docker Hub is a public repository of Docker images that can be downloaded with *docker pull* command over HTTP.
The score assigned to Docker for the availability is 4/5, given the need to be installed in most distributions.

Functionality. Docker is an engine which is focused on providing an execution environment for one process only. This means that in general there should be a 1:1 ratio between Docker containers and applications running and also that a container terminates when the process with PID 1 exits.
Docker offers a rich set of functionalities to build, run and manage containers as it is described below.
One of the main features of Docker that distinguishes it from other container engines is the way it handles images. The building process for Docker images is composed of different elements playing various roles; a Dockerfile which is a text file that describes the instructions to build the image. Each instruction in a Dockerfile results in a new image layer. Every new layer is then created running the previous one, executing the next Dockerfile instruction, and saving the result [66, p.40-45]. It is possible to build an image starting from an arbitrary image as a base.
It is possible to share files with the host through Docker volumes. This can be achieved both at building time, specifying the directories in the Dockerfile, or at runtime.
By default, when no arguments or options are specified, Docker sends everything logged to STDOUT or STDERR (Standard Output and Standard Error). The logs can then be retrieved with *docker logs* command or through the HTTP API.
It is possible to collect information about events involving containers such as when a container is created, destroyed, restarted and so on [66, p.188-189].
Given the extensive functionalities that Docker offers, a score of 5/5 is assigned.

Performance. One execution of the benchmark script in a Docker container takes on average 3.5s. The execution time is in line with the other platforms and it is lower than the native execution time, therefore a score of 5/5 is assigned.

5.1.3 LXC/LXD

LXC is defined as "a userspace interface for the Linux kernel containment features" [8] while LXD is just a manager tool built on top of LXC [9].
Unlike Docker, the purpose of LXC is to provide a full virtual environment without the burden of running a separate kernel and a hypervisor. Thus, in this case, several applications can run inside one singe container.
The summary of the scores assigned to LXC can be found in Table 8 below.

Table 8. Summary of the scores assigned to LXC container engine.

Parameter	**Score**
Availability	3
Functionality	3
Performance	4

Availability. There are few hard dependencies for LXC, including a C library (like Glibc) and the kernel version 2.6.32 or superior. These requirements seem to be likely fulfilled on most systems and therefore do not impose a hard constraint. Installing LXC/LXD is generally possible through the package manager, but even the most supported operating system -Ubuntu- does not come with LXC preinstalled. It is possible to import public "templates" for different operating systems, but there is not any public repository for already made containers. It is still possible to distribute images via HTTP with a private webserver.
LXC requires a number of dependencies to be satisfied, is not preinstalled on any distribution and there is no public image distributions. For this reasons, a score of 3/5 is assigned in terms of availability.

Functionality. The functionalities offered by LXC are many and powerful, but in general require careful configuration. For example to have a container with Internet access it is necessary to configure a bridge in the host side, while for both docker and rkt, this is performed automatically and containers have Internet access "out-of-the-box".
The preparation of an image differs quite a lot from the Docker building process. With LXC it is possible to "export" and then "import" an image. The export produces usually a tar.xz

file or two tarballs (in which the metadata and the file-system are split) that can be imported. The exported image in this case is more similar to a snapshot of a running container than to a fresh image.
It is possible to share part of the file-system with the host by simply mounting it inside the container, provided a correct permission scheme.
Given the different design of LXC/LXD, the logs of applications running inside the container must be extracted directly from inside the container but it is also possible to specify a file on which to log the STDOUT and STDERR of the container, similarly to Docker.
The events such as creation and exit of containers are present in the host's system log. It is important to keep in mind that LXC containers are not as disposable as Docker ones, therefore it is reasonable not to have a more abstract way to monitor container events.
LXC offers most of the functionalities needed, but requires a much heavier configuration and allows a reduced flexibility, therefore a score of 3/5 is assigned for its functionalities.

Performance. One execution of the benchmark script in a LXC container takes on average 3.9s.
The execution time is slightly slower than in the other platforms and also slower ($< 10\%$ overhead) compared to the native machine, therefore a score of 4/5 is assigned.

5.1.4 Rkt

Rkt [15], read "rocket", is a container engine developer by CoreOS Container Linux team, and -apart from low-level details- it is very similar to Docker. Like Docker and in opposition to LXC, rkt containers are meant to run a single application to which their life depends on.
The summary of the scores assigned to rkt can be found in Table 9 below.

Table 9. Summary of the scores assigned to rkt container engine.

Parameter	**Score**
Availability	4
Functionality	4
Performance	5

Availability. Rkt is available from the package manager in most Linux distributions, with the exception of Ubuntu and CentOS, and it is preinstalled on CoreOS Container Linux [13]. The only dependency for running rkt containers is a kernel version 3.18 or later.

Rkt can run both Application Container Images (ACI) or Docker containers. For the first, the location of the container needs to be known a priori to be downloaded, while for the latter it is possible to use the docker hub. There is not an official rkt repository similar to Docker Hub.

Despite the ease of installation, the lack of a native public repository for images and of native support for such major distributions entail a score of 4/5.

Functionality. The functionalities of rkt, especially from the perspective of this project, are very similar to the ones offered by Docker, except for building a new image.
Since rkt can run Docker images, it is possible to use Docker build feature, but in a more general process, the build of a new container needs to be done through *acbuild* [1] tool, that appears to be unmaintained.
For what concerns sharing files or directories with the host, rkt uses volumes in a very similar fashion to Docker, allowing to mount a volume both at runtime or at build time.
It is possible to collect logs from containers via *journalctl* from the host, except for those applications which by default log to */dev/stdout* and */dev/stderr*, in which cases it is necessary to specify the log destination manually [14].
Finally, similarly to Docker, it is possible to monitor the events, such as creation and termination, of a container with the rkt CLI (Command Line Interface).

The functionalities of rkt are very similar to the ones of docker, with the exception of the building process which leads to a score of 4/5.

Performance. One execution of the benchmark script in a rkt container takes on average 3.33s.
The execution time is in line with the other platforms and it is lower than the native execution time, therefore a score of 5/5 is assigned.

5.1.5 Conclusion

Taking into account the scores and the considerations made in the previous sections, the results are described in Table 10.
From the analysis performed Docker results the most suitable candidate and it will be the platform used in this project.
It is important to remember that this evaluation is meaningful in the context of this project, while opposite results could be obtained for the same evaluation with a different use case. For example, to deploy complex applications, LXC seems a more valid choice than Docker or Rkt

Table 10. Final results for the container engines evaluation.

Container Engine	Availability	Functionality	Performance	Total
Docker	4	5	5	47
LXC	3	3	4	32.5
Rkt	4	4	5	42.5

which are oriented toward single processes, and the price of a heavier configuration in that case could be worth the result of having all the tools and components in one system, similarly to a virtual machine. In the case of this research however, there is no required interaction between the tools used and therefore the orientation of both Docker and rkt toward a single-process container is suitable.

5.2 Programming language

The project is developed using *Python* language. The main advantage of using Python over other possible languages (C, among the others) is a library [6] that implements all the operations normally available through the *docker* command or API.

5.3 Choosing defense techniques and tools

The choice of the defense techniques to verify represents the functional core of the project. This selection needs to be informed and to represent a compromise between reasonable security and likelihood for a security measure to be applied to a common system. In other words, it is unproductive to produce a tool that expects on a system all the tools described in section 3.6, creating an unnecessary number of red flags for which it is very likely that no action will be taken.
In the description below the rationale behind the choice of each defense measure will be presented while Table 11 shows a summary of the choices made and a brief description of the reason for it.

Libsafe and derived. Libsafe or similar derived tools will not be taken into account and TED will not attempt to verify that they are in place. These, in fact, are relatively outdated and invasive defense techniques. As it has been discussed, it is necessary to replace some native Glibc functions (some of which have even been secured) in order to apply these measure and also Libsafe-like techniques present severe limitations in terms of efficacy. Another reason

Table 11. Summary of the choice for the defense techniques to verify and their motivations.

Name	Chosen	Brief reason
Libsafe and derived		Relatively invasive, limited efficacy, outdated and not used.
SSP	X	Commonly used, simple but effective in its scope.
NX	X	Commonly used, effective against a number of attacks.
ASLR	X	Very important measure, commonly used, effective.
AAAS		Replaced by ASLR, outdated.
Safe Unlink		Limited efficacy, very basic measure.
Stripping	X	Simple and common measure, although not very effective.
KASLR/KPTI	X	Effective against Meltdown, recent.

not to consider these protections is that the range of attacks they protect from overlaps with other more important protections such as NX.

Stack Smashing Protector and canaries. Stack Smashing Protector will be considered, and TED will expect that each binary will be compiled using canaries. The reason for this is that GCC, the standard GNU C Compiler, uses SSP as a default for several versions. This means that to have a program compiled without SSP it is necessary to use some special flags during compilation (*-fno-stack-protector*) and therefore, if this is done, there should be a reason for it. Moreover, not only SSP is very common thanks to the integration into GCC, but it provides also a good level of protection compared to the performance and complexity overhead.
Other solutions such as shadow stack based defenses ([78] [90]) will not be taken into consideration because there is a reasonable doubt that they are commonly used.
To detect whether a binary is protected with SSP, both *objdump*[1] from *binutils* and *rabin2* [2], part of *radare2* [3] framework, are used.

Non executable pages (NX). NX protection will be considered, and TED will expect that each binary will not contain data sections marked as executable. TED targets explicitly modern systems, which in the vast majority of cases run 64-bit architecture; 64-bit CPUs support natively the *NX* bit and GCC by default compiles marking the stack non executable. In fact, a binary compiled with default settings with GCC will be also compliant to W^X, and therefore TED will not only verify that the stack is not executable, but also that there are no sections which are marked both writable and executable. Although PaX enforces the NX at a lower level, by restricting *mmap()* calls for example, and therefore increases the level of security,

[1]`https://linux.die.net/man/1/objdump`
[2]`https://radare.gitbooks.io/radare2book/content/rabin2/intro.html`
[3]`https://github.com/radare/radare2`

there is no available data concerning the usage of PaX besides the fact that this is included by default in some minor Linux versions, and thus the PaX verification will not be included in TED, in order to avoid a large number of false positives. However, TED verifies NX and Wˆ directly on each binary, putting in evidence cases in which these are not respected. The tool chosen to verify NX and WˆX is the already mentioned *rabin2*.

Address Space Layout Randomization. ASLR will be considered in this project, and TED will verify if ASLR is present in the system and to what extent is enabled. The address space randomization is considered in this project one of the most important security measures since it helps mitigating many different types of attacks. From the information gathered in section 3.6.4 it is possible to notice that ASLR is not very effective on 32-bit architectures, but since this project focuses on AMD64 architectures, most of the concerns raised do not apply. TED will therefore verify that ASLR is active on the system, and will also detect whether additional security patches are applied to improve its efficiency. However, in this project other, more advanced, projects such as those mentioned in section 3.6.4 will not be taken in consideration both because of the complexity of such verification and the unlikelihood for them to be in use. The tools used to implement the ASLR verification are the standard *sysctl* binary [1] and a custom written binary, whose code can be found in Appendix 1 on page 97.

ASCII Armored Address Space. AAAS will not be taken into consideration in this project. This security measure is outdated, not very effective and several ways to bypass it exist. Besides, with ASLR enabled the address of libraries should be randomized, and imposing the constraint of placing these libraries in the armor zone would weaken the security of ASLR for little to no benefit.

Safe unlink macro and heal protection. The safety of the unlink macro will not be taken into account by TED. The reason for this lies on how this would improve the system security. As it has been discussed in section 3.6.6, the unlink macro has been patched long ago and is now included in the default Glibc. Despite this, developers have the freedom to choose which heap allocator to use, they do not have to rely on the system's *malloc()* and therefore the fact that system's Glibc is up to date and the unlink macro is safe does not mean that the compiled binaries are safe from heap attacks. Besides, the unlink attack is the simplest and oldest heap overflow attack, while several others have been developed that are not mitigated by this code fix. Moreover, there is not reasonable evidence that the advanced solutions

[1] `https://linux.die.net/man/8/sysctl`

discussed in section 3.6.6 are used or common, and therefore TED will not attempt to verify their employment, leaving ASLR as the only protection against heap overflow attacks.

Stripping binary. The stripping of binaries will be taken into account in this project. TED will determine if each binary is stripped. This is not only a common anti-reverse engineering technique which is used, but it is almost the only one available in Linux environments. This measure offers very little security in terms of countering reverse engineering, and therefore TED will consider it giving it the proper weight. Despite this, the binaries shipped with Linux distributions are in general stripped, and if a custom binary is distributed with valuable information, it would be a good practice to strip it as well. Because of this, it will be up to the specialist performing the scan to understand whether the fact that a binary is stripped or not constitutes a threat (for example, an unknown binary is stripped) or a lack of security (for example, a known important binary is not stripped). The tool used to implement the stripping verification is *rabin2*.

KAISER/KPTI. KAISER and KPTI will be considered but not verified directly in this project. Instead, an external tool will be used to verify the vulnerability of the system against Spectre and Meltdown attacks, and this will therefore indirectly verify whether KPTI is enabled on the system.

System vulnerability. As it has been mentioned in section 4, in addition to the verification of the defense measures, a basic vulnerability scan is performed on the system. This is composed by two checks:

1. The first check verifies whether the kernel is vulnerable to known exploits. The tool used to perform this check is a slightly modified version of *Kernelpop*[1]

2. The second check verifies whether the system is vulnerable to three different variants of Spectre attack. The tool used to perform this check is a script called *spectre-meltdown-checker.sh* [2]

5.4 Scoring system

In order to facilitate the vulnerability assessment a score is assigned to the system and another is assigned to each binary. In the result TED will present the single score for each test and

[1]`https://github.com/spencerdodd/kernelpop`
[2]`https://github.com/speed47/spectre-meltdown-checker`

the aggregated scores. The scoring system is established in the methodology and presented in section 4.2.1.

5.5 Implementation details

In this section the practical aspects of TED's implementations will be discussed. First, a general overview of the structure of the tool will be presented. In this overview, elements such as the general design, the flow of the code and the high level logic will be analyzed. The second part will instead describe how the verification of the techniques chosen in section 5.3 has been implemented with the aid of the mentioned tools and some details of the code will be discussed.
The code of TED is publicly available and can be found on a public Github repository[1]. All the Docker images are available either on the public Docker Hub [2] or on the author's Docker Hub page[3].

5.5.1 General design

The program is composed by a main script called *ted.py* and by several classes that can be found in the *src* directory, each of which implements a different check or provides some supporting functionality (output parsing, data collection etc.).

Main script functionalities. The main script accepts several parameters: the type of the scan, the output format, the output report filename and the path to scan.
TED defines three different types of scan that can be performed:

1. Full scan, option *-f*. This scan includes system wide checks and the test of all the binaries present in the path specified.

2. Elf only scan, option *-e*. This scan does not perform system wide checks and directly scans all the binaries present in the path specified.

3. Targeted scan, option *-t*. This scan performs the scan of a specific binary whose path has been specified.

TED can produce output in JSON format and in CSV format, although more functionalities are available by using JSON format at the moment.

[1]`https://github.com/Sudneo/TED`
[2]`https://hub.docker.com/`
[3] `https://hub.docker.com/u/sudneo/`

Main script flow. The code goes through several logical steps:

1. Determine the type of the requested scan depending on the parameters passed by the user.
2. If the type of scan is "full", then execute the system verifications one by one and collect the results.
3. If the type of scan requested is "elf only" or "full", then find all binaries in the path specified and analyze them, collecting the results.
4. If the type of scan is "targeted", then analyze the target binary and collect the result.
5. After the results from each test performed are collected, compute the scores.
6. Finally, produce the report which includes the findings.

Code organization. By design, every specific function is implemented in its own class in a file placed in the *src* directory. A brief overview of the files in this directory and a brief description of their purpose can be found in Table 12.

Table 12. Summary of files present in src directory and their purpose.

File	Desription
aslr_scan.py	Class used to detect the ASLR level.
elf_finder.py	Class used to collect of the binaries in a path.
elf_scan.py	Class used to gather information on a binary.
kernelpop_scan.py	Class used to perform the system scan with Kernelpop.
nx_scan.py	Class used to verify the NX support from the CPU.
output_parser.py	Class used to parse the outputs from the various tests and produce a report.
scorer.py	Class used to assign the scores to each test.
single_elf_scanner.py	Class used to determine the risk for a binary given the information gathered.
spectre_meltdown_scan.py	Class used to perform the system scan with *spectre-meltdown-checker.sh*

Docker images. As it has been discussed, all the tests are performed inside Docker containers, leaving just supporting functionalities (output parsing, scoring, production of reports) to be performed on the native machine. In order to develop TED it has therefore been necessary to prepare and build several Docker images. A summary of the images built can be found in Table 13.

Table 13. Summary of Docker images built and their brief description.

Image	Desription
sudneo/kernelpop	Image containing an installation of Kernelpop tool.
sudneo/radare2	Image containing an installation of radare2 framework
sudneo/aslr_check	Image containing a custom binary to verify the ASLR configuration.
sudneo/spectre-meltdown	Image containing the spectre-meltdown-checker.sh script.

Report production and parsing. TED can produce reports both in JSON and CSV format. While CSV format might be convenient to sort data easily and have a shorter summary, the most supported format is JSON. In order to provide a way to have a historical view of the system, the standard *diff*[1] tool is not convenient since it is not aware of the logic of TED reports. To solve this problem, a custom tool TEDiff has been developed an its code is publicly available[2]. This tool allows to compute the differences in reports in terms of binaries added/removed from the system, changes in system vulnerabilities and changes in each of the details of every single elf.

5.5.2 Implementation specifics

In this section the technical details of the Docker images used and of the classes that implement the tests are discussed. The details concerning the supporting classes will not be discussed since not relevant with the general context of this project.
The logical flow of the program is shown in Figure 6, where the blocks colored in yellow are executed inside Docker containers, while those in blue are executed on the native machine.

[1]`https://linux.die.net/man/1/diff`
[2]`https://github.com/Sudneo/TEDiff`

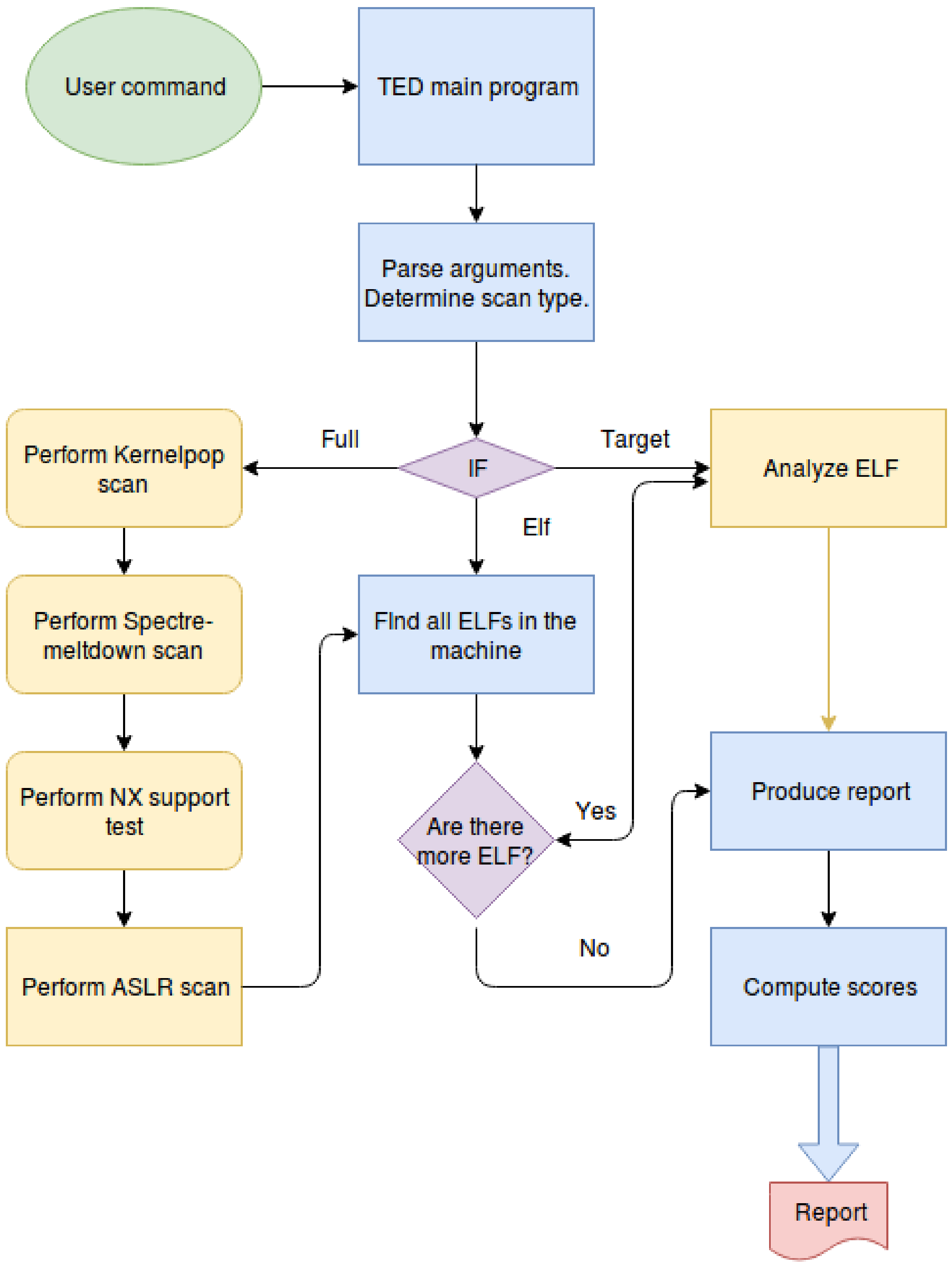


Figure 6. Flow chart of TED execution.

Docker images. The following are the Docker images produced to support TED and their specifics:

1. sudneo/aslr_check
   (a) Base image: debian:8
   (b) Files added: *get_addresses*.
   (c) Programs installed: -
   (d) Purpose: This image is used to get the ASLR configuration in the system.
2. sudneo/kernelpop
   (a) Base image: debian:8
   (b) Files added: *entrypoint.sh*, checkout of the custom *Kernelpop* branch[1]
   (c) Programs installed: *Python3*, *wget*, *unzip*, *git*
   (d) Purpose: This image is used to run a customized version of Kernelpop on the system.
3. sudneo/radare2
   (a) Base image: debian: latest
   (b) Files added: -
   (c) Programs installed: *radare2*, *binutils*
   (d) Purpose: This image is used to provide an interface to run *rabin2* and *objdump* commands.
4. sudneo/spectre-meltdown
   (a) Base image: debian:8
   (b) Files added: -
   (c) Programs installed: *wget*, *binutils*, *spectre-meltdown-checker.sh*
   (d) Purpose: This image is used to run the spectre-meltdown-checker script against the system.

ted.py. The main script implements the orchestration of the operations to perform, such as collecting arguments from the users, scheduling the scans, building and writing the report. The collection of the arguments happens through the *argparse* Python package.

[1]`https://github.com/Sudneo/kernelpop`

aslr_scan.py. This class implements the verification of ASLR in the system. To do so, the sudneo/aslr_check Docker image is used. In this context, two different tests are performed: a soft scan and a hard scan. The soft scan consists in running the *sysctl kernel.randomize_va_space* command. The output can be a number from 0 to 4.

1. **0** means ASLR is not enabled
2. **1** means ASLR is partially enabled
3. **2** means ASLR is enabled
4. **3** or above means ASLR is enabled and further patches are applied

For reference it is possible to compare this list with the description made in section 5.4.
The hard scan consists in an iterative test using the *get_addresses* binary present in the image (cfr. Appendix 1 page 97). When this binary is run it prints the address of the stack, the address returned by a *malloc()* call and the address of the environment variables. This script is run 10 times and for each execution the addresses are collected in three different sets. If a repetition of an address is found in the set containing the stack addresses or the environment variables, then it is assumed that ASLR is disabled (0). If a repetition is detected in the set containing heap addresses but not in the others, then ASLR is considered enabled partially (1). Finally, if no repetitions are found, then ASLR is most likely fully enabled (2 or more), but the full extend cannot be determined, for example it cannot be determined whether hardening patches are applied.
After both the soft and the hard scan are run, the result is prepared in a JSON object and returned to the caller.

elf_finder.py. Despite this class does not implement specifically a security check, it provides the very important function of collecting all the binaries in a given path. This means that indirectly the code in this class influences the content of the reports produced and therefore it is worth discussing its working mechanism and limitations. In order to traverse the tree path starting from a given point (the path chosen), the *walk* function in *os* Python package is used. This function allows to iterate also on hidden files and directories, therefore it is very suitable to recursively explore a given path. The determination of ELF files is performed by reading the first 4 bytes of the file and comparing it with the hexadecimal string *7f454c46*, which corresponds to the ELF "magic number" [34, p.1-7].

Limitations. In the ELF finder there is a limitation needed to have a working proof of concept and that is meant to be resolved in the future. This limitation is that the operation of reading from file (to check the magic number) is limited to a maximum duration of 1 second. The reason for this is that in Unix everything is a file, including devices and interfaces and therefore attempting to read on such files might stall the execution for an indefinite amount of time. The value of 1 second is established as a reasonable upper-bound of the read operation.

elf_scan.py and single_elf_scanner.py. These two classes work closely together in providing the security information about a given ELF file. The single_elf_scanner class consumes and parses the output produced by elf_scan and produces as result the report for the binary analyzed. In the elf_scan the sudneo/radare2 Docker container is used to perform three main operations:

1. Run rabin2 -Sj $ELF. This command prints the section information of the ELF file passed as parameter.
2. Run rabin2 -Ij $ELF. This command prints general information about the ELF file passed as parameter.
3. Run objdump -d $ELF. This commands prints the disassembled version of the ELF file passed as parameter.

Once the information from these commands is gathered, the code in single_elf_scanner determines the result of the tests as described below.
The binary compliance to NX is verified by three sub-operations:

1. Checking whether the general information about the binary contains the NX flag. (NX test)
2. Checking whether the stack section contains the executable flag. (Executable stack test)
3. Looping over the section information of the binary and checking whether any section contains both the executable and writable flag. (W^X test)

The usage of canaries is verified by two sub-operations:

1. Checking whether the general information about the binary contains the "*canaries*" flag. (canaries test)
2. Checking whether the disassembled code of the binary contains the string "__stack_chk_fail". This is the name of the function that verifies the integrity of the canary in the GCC SSP implementation. (SSP test)

The binary is considered stripped if the general information report about the binary contains the "*stripped*" flag.
Finally, the single_elf_scanner code prepares and returns the report for this specific binary in a JSON object including the results of all the tests and the SHA256 hash of the file.

kernelpop_scan.py. This class implements the scanning of the system with *kernelpop*. The logic of this class is very easy, since the Docker container used sudneo/kernelpop has already kernelpop installed inside, specifically, a modified branch that is able to write its report on a file. After having started the container, the command *Python3 /kernelpop/kernelpop.py -r report.csv* is executed by the *entrypoint.sh* script. Once the execution has terminated, the report is collected and each confirmed or potential exploit found is added to a JSON object that is returned to the caller.

nx_scan.py. This class implements the verification of the support for the NX bit by the processor. This class uses a public Docker container, specifically the debian:latest container to run a simple command: *cat /proc/cpuinfo*. This command prints -among the other information- a list of flags that indicate the support from the CPU. If "nx" or "pae" are among these flags, then the CPU supports the NX bit natively or, in case of pae, the address extension to support it. The result of this test is inserted in a JSON object and returned to the caller.

spectre_meltdown_scan.py. This class implements the scanning of the system with *spectre-meltdown-checker.sh*. In this class the Docker container sudneo/spectre-meltdown is used, which contains inside the script ready to be executed. The following command is executed, *bash /spectre-meltdown/spectre-meltdown-checker.sh –batch json*, which runs the script and instruments it to produce the output in JSON format. The JSON object returned from the script is then passed back to the caller.

6 Testing and validation of the results

In this section the design, the practical implementations and the results of the testing and validating process will be discussed. Several aspects needed to be tested and validated to answer the following questions:

1. Is TED working?
2. Does TED provide the functionality requested, does it allow to perform a risk assessment on the binaries of a system?
3. What is the trade-off of using a containerized application?

In order to answer all these questions, initially two types of tests have been performed, consistently with the methodology established in section 4.4. The first test is a functional test, which consists of running TED on several machines before and after a controlled tampering, analyzing the result produced and comparing them with the expected results. This test and its result will be discussed in section 6.1. The second test is a benchmark in which the same operations have been executed on a native machine and inside a Docker container, in order to measure the overhead represented by the containerization. This test and its outcome will be discussed in section 6.2.
The machine used to perform these tests has the specifics described in Table 14 below.

Table 14. Specifics of the machine on which the functional and performance tests have been performed.

Model	Lenovo Y50-70
Memory	8GB RAM DDR3 1600Mhz
CPU	Intel i7-4710HQ@2.50Ghz
Hard disk	ATA disk WD10SPCX-24H 5400 rpm

Once the tests in the controlled environment have been performed, to further prove the effectiveness of TED, and according to the research methodology established, two practical real world tests will be executed. The first is a functional test, consisting on running TED on a public server, examine the results and verify whether TED allows to individuate any real critical issues and allows to determine hardening actions that need to be taken. If such critical issues are found, then the server will be hardened and TED will be run again. After this second step, if TED shows that the critical issues have been resolved, then the test can be considered successful. The second test consists in running TED on a portion of the Locked Shields infrastructure for blue teams and verifying that TED individuates vulnerabilities and eventual planted binaries.

6.1 Functional test

The functional tests is a qualitative test that aims to answer to questions 1 and 2 mentioned in section 6.
The test is performed according to the methodology established in section 4.4.
The three different operating systems chosen are Debian 9.3.0 Wheezy, Ubuntu 16.04.4 server and openSUSE Leap 42.3. These operating systems have been chosen because they are three of the most popular server Linux distributions [36].
A summary of the functionality evaluation is shown in Table 15.

Table 15. Summary of the functionality test for the three Virtual Machines tested.

Platform	Dep.s satisfied (/7)	Slowest TED execution	Tampers detected (/4)
Debian 9.3.0	3/7	46m15s	4/4
Ubuntu 16.04	4/7	51m36s	4/4
openSuse 42.3	4/7	66m49s	4/4

6.1.1 Dependencies

One of the requirements of TED is the portability and specifically a reduced number of dependencies, therefore discussing the required ones for TED is a necessary step to verify TED's functionalities.
TED has the following dependencies:

1. Python 2.7
2. stopit Python package
3. docker Python package
4. docker
5. wget and unzip (or git) in order to get TED's code.
6. An Internet connection

Table 16 summarizes the dependency satisfaction in the fresh installed systems.
As expected it is necessary to install Docker container engine in all the three servers. The procedure for this is quite straightforward and required only *git* to be installed as additional dependency. In order to install the Python packages, it is possible to simply download them

Table 16. TED's dependencies summary for the three fresh installed systems used for the functional test.

Dependency	Debian 9	Ubuntu 16.04	suse 42.3
Python 2.7	X	X	X
stopit Python package			
docker Python package			
docker			
wget	X	X	X
unzip		X	X
Internet connection	X	X	X

and manually install, but for a matter of simplicity in this context *pip* has been used and therefore installed on the system. Finally, the Debian machine required to install also *unzip* binary, although it was possible to install *git* to get the TED code, since this is anyway a dependency for Docker.
Considering the relatively small amount of dependencies, the ease of their satisfaction, the number of tools that these dependencies will allow to run and the fact that it is possible to package the application removing the need for Python packages installations (hence reducing even more the number of dependencies) it is possible to conclude that TED fits the requirement of reducing the dependencies on the system and can speed up the deployment of the tools used. It is also worth mentioning that in the future a TED container will be built, making Docker the only dependency. This means that on Linux cloud distributions, where Docker is commonly installed by default, TED will have no dependencies at all.

Executing TED in special environments. In some cases, especially in CSIRTs operations, it might not be possible to install new packages, modify the drives content or have an Internet connection available. Executing TED in such scenarios is harder, but it might still be possible. The Internet connection is necessary exclusively to download the Docker images from the public repository, but it is also possible to pack them together with TED source code to overcome the lack of a connection, gaining the ability to run the tool in offline or air-gapped machines. Running TED without the ability to install new packages can be made possible in a similar way by packing Docker binaries and Python packages together with the application. Finally, TED at the moment has not the ability to run without modifying the disks; specifically, simply running a default Docker container modifies the content of the */var/lib* directory. Despite this, TED does minimal use of temporary files and mostly works in memory, therefore one of the possible future works can be developing TED with the ability to run in scenarios with heavy limitations, for example by running it from a USB (Universal Serial

Bus) drive without modifying any other drive.

6.1.2 Functionalities

The main purpose of TED is to provide a security risk assessment of a system's binaries. This goal is reached if by running TED it is possible to obtain a report that allows to individuate critical issues or potentially vulnerable files. In order to verify the achievement of this goal, TED has been run on each of the three machines installed as soon as its dependencies have been satisfied, following the methodology proposed. After this first run, each server has undergone a tampering simulation, running a custom written script, with the purpose of including unprotected binaries in the system. The code for this script is listed in Appendix 2 on page 98. Once the tampering (whose specifics is described below) is finished, TED is run again and the differences between the two reports are analyzed.

Tampering the server. The tampering of each machine consists of downloading and installing in the system vulnerable binaries. This is achieved with a simple bash script that does the following:

1. Downloads the source code of the *coreutils* Linux package.
2. Compiles each file in this package using *fno-stack-protector* and *-z execstack* flags.
3. Overwrites with the binaries obtained the original binaries in the */bin* directory.
4. Downloads a vulnerable C source code file[1].
5. Compiles this file with the *fno-stack-protector* flag.
6. Picks a random binary in the */sbin* directory and overwrites this with the obtained binary.
7. Overwrites the *kernel.randomize_va_space* kernel setting with a number among 0 and 1, disabling or partially disabling ASLR.
8. Cleans the downloaded files.

[1]`https://raw.githubusercontent.com/npapernot/buffer-overflow-attack/master/stack.c`

Running the tool. The following commands have been used to run the TED scan, assuming that the dependencies have already been statisfied and that $MACHINE is for example the name of the machine.

1. *wget https://github.com/Sudneo/TED/archive/master.zip*
2. *unzip master.zip*
3. *cd TED-master*
4. *python ted.py -f -Oj $MACHINE.json -p /*

The report can be retrieved from $MACHINE.json.

Expected results. Given the testing and tampering process followed, comparing the TED reports from the first and the second (after the tampering) run the following differences should be present:

1. One binary in the */sbin* folder should have a different score.
2. Many binaries in the */bin* folder should have a different score.
3. New binaries should be added to the system (for example the object files result of the compilation of *coreutils*) and no binaries should be removed.
4. The system score, specifically related to ASLR, should have changed.

6.1.3 Result

Analyzing the differences in the reports between the fresh and the post-tampering scan it has been possible to observe the following results:

1. In all three cases the different ASLR configuration has been detected and reported, increasing the system score by the corresponding points.
2. In all three cases one binary in the */sbin* directory has been marked as changed and the differences (including the SHA256 hash) reported.
3. In all three cases several binaries in */bin* have been marked has changed and their security score has been updated.
4. In all three cases no removed binaries have been registered.
5. In all three cases several binaries have been reported as added to the system.

With the results obtained it is possible to conclude that TED successfully provides a relatively easy and independent solution for an effective security risk assessment and continuous monitoring of binaries present on the system. The solution requires minimal dependencies, is self-contained and provides an effective way to have a system overview in relation to binary security.

6.2 Performance test

The performance test is a quantitative analysis with the aim of measuring the impact of running tests inside containers and of deciding if this approach is feasible in the real world, therefore this analysis allows to establish the trade-off between simplicity and performance. In order to perform an accurate but also relevant test for this project and in accordance with the methodology established in section 4.4, the ASLR check, the Spectre check, the Kernelpop check and the analysis of an ELF file have been executed. Each test has been executed separately and the time needed for its execution has been measured with *timeit*[1] Python package, meaning that the times reported are the ones needed to execute the function that implements the test.

ASLR. The ASLR test has been executed 100 times inside a the *sudneo/aslr_check* Docker container and 100 times on the physical machine.
The summary result is described in Table 17.

Table 17. Summary result for the benchmark test of the ASLR check.

	Physical	**Docker**	**Relative Δ**	**Absolute Δ**
Average	0.031s	0.469s	+1418%	0.438s
Minimum	0.021s	0.412s	+2084%	0.427s
Maximum	0.039s	0.555s	+1105%	0.429s

On average performing the ASLR test inside a container took about 0.44s more than executing it on the physical machine.

Kernelpop. The kernelpop test has been executed 100 times inside the *sudneo/kernelpop* Docker container and 100 times on the physical machine.
The summary result is described in Table 18.
On average performing the Kernelpop test inside a container took 1.44s more than executing it on the physical machine.

[1] `https://docs.python.org/2/library/timeit.html`

Table 18. Summary result for the benchmark test of the Kernelpop check.

	Physical	**Docker**	**Relative ∆**	**Absolute ∆**
Average	0.112s	1.559s	+1284%	1.447s
Minimum	0.094s	1.073s	+1033%	0.978s
Maximum	0.175s	1.857s	+959%	1.681s

Spectre. The Spectre test has been executed 100 times inside the sudneo/spectre-meltdown Docker container and 100 times on the physical machine.
The summary result is described in Table 19.

Table 19. Summary result for the benchmark test of the Spectre check.

	Physical	**Docker**	**Relative ∆**	**Absolute ∆**
Average	4.745s	6.066s	+27.8%	1.320s
Minimum	4.595s	5.750s	+25.1%	1.153s
Maximum	5.003s	7.366s	+47.2%	2.362s

On average performing the Spectre test inside a container took 1.32s more than executing it on the physical machine.

ELFs. The ELF test has been executed by scanning all the binaries in the */bin* folder (130 non symlink binaries) and running the related ELF checks inside the *sudneo/radare2* Docker container and on the physical machine. The execution has been measured for each single binary rather than for the whole execution.
The summary result is described in Table 20.

Table 20. Summary result for the benchmark test of the ELF check.

	Physical	**Docker**	**Relative ∆**	**Absolute ∆**
Average	0.00060s	0.00055s	-8.4%	0.00005s
Minimum	0.0002s	0.0002s	0%	0s
Maximum	0.0057s	0.0053s	-7.0%	0.0004s

On average performing a check of an ELF file inside a container took 0.5ms less than executing it on the physical machine.

6.2.1 Observations

All the benchmarks presented show a similar behavior, except the ELF one in which executing commands inside the Docker containers results faster than the corresponding commands on the physical machine. The reason for this difference can be understood by individuating what slows down the execution in Docker in the other scenarios. For system checks, such as ASLR, Kernelpop and Spectre, the test is executed once and it requires the creation (and termination) of the container. For the ELF scan instead, a container is started, but then all the binaries that need to be scanned are scanned inside this same container. It is clear therefore that the performance loss derives from the bootstrap of the Docker container. To confirm this claim, the Kernelpop and Spectre test have been run 100 times inside a single Docker container, without requiring the creation for each test. The results obtained are described in Table 21.

Table 21. Result of Kernelpop and Spectre test without the bootstrap of container at each execution.

	Physical	**Docker**	**Docker no startup**	**Relative Δ**	**Absolute Δ**
Avg. Kernelpop	0.112s	1.559s	0.134s	+16.4%	0.022s
Min. Kernelpop	0.094s	1.073s	0.126s	+25.2%	0.032s
Max. Kernelpop	0.175s	1.857s	0.140s	-25.4%	0.034s
Avg. Spectre	4.745s	6.066s	4.485s	-5.8%	0.260s
Min. Spectre	4.595s	5.750s	4.365s	-5.3%	0.230s
Max. Spectre	5.003s	7.366s	5.212s	+4.0%	0.209s

From this test is clear that without the startup of the container the execution time becomes almost the same, being faster to execute tests inside containers in some cases.

6.2.2 Result

The result of the performance benchmark shows that the overhead in running tests inside containers is noticeable in relative terms, but very minimal in absolute terms. The biggest portion of the execution time of a TED scan is spent scanning all the different binaries, and this operation is performed in a single container and therefore without any performance loss (the test showed better results inside containers). The operations that have some overhead are executed only once, and cannot be considered a serious penalization. In fact, the total execution time for a full TED scan ranged from 46 to 66 minutes, as reported in Table 15 on page 74, and a combined overhead of 10s in these cases would represent an increase of 0.36%-0.25%, respectively.

If performances are a concern however, it is also possible to merge all the Docker containers used inside one single container that has all the required tools/binaries installed, similarly to what would have happened using LXC has container engine. This approach though is against the design of Docker and given the limited benefits, it is not recommended. Considering all the observations and results, it is possible to conclude that running tools inside containers effectively removes dependencies, speed up the deployment at the price of a negligible performance overhead, and therefore can represent an efficient solution for security frameworks in the future.

6.3 Real world test

This section contains the description of the execution of the two practical real world tests mentioned in section 6 and established in the methodology in section 4.4.

6.3.1 Public server.

The first test consists of executing TED on a public server and examining the results. The server on which TED has been executed is hosted by Online.net[1] and has the characteristics summarized in Table 22.

Table 22. Specifics of the server on which the first real world test has been performed.

Memory	3947MiB DDR3
CPU	Intel(R) Atom(TM) CPU C2338 @ 1.74GHz
Hard disk	SanDisk X400 2.5 SSD
Operating system	Ubuntu 16.04.3 LTS
Kernel version	Linux 4.4.0-97-generic x86_64
Dependencies needed	Docker, *stopit* and *docker* Python packages.

TED full scan took 84m2s to complete. The issues found are summarized in Table 23 together with the actions that will be taken to mitigate them. The redacted report for this scan is used as an example and it is shown in Appendix 3 on page 99.

Observations. Table 23 shows that all the actions defined involve the system and the kernel, while no actions are taken for the ELFs compiled without defenses. The reason for this is multi-faced; first, most of the binaries, if not all, are used for local commands and are not facing the public, meaning that their exploitation could lead to local access, which is already necessary to exploit them on the first place. Second, none of the binaries have the *setuid*

[1] `https://www.online.net/en`

Table 23. Specifics of the server on which the first real world test has been performed.

Type	**Desription**	**Action**
System	Kernel vulnerable to dirtyCOW	Upgrade kernel
System	Kernel vulnerable to dirtyCOW poke variant	Upgrade kernel
System	Kernel vulnerable to CVE20177308[1]	Disable user namespace usage to unprivileged users.
System	Kernel vulnerable to CVE20162384[2]	Update kernel or restrict USB access.
System	Kernel vulnerable to CVE20176074[3]	Disable kernel DCCP module.
System	Machine vulnerable to Spectre variant 1	Upgrade kernel
System	Machine vulnerable to Spectre variant 2	Upgrade kernel
System	Machine vulnerable to Meltdown	Enable KPTI
ELFs	71 executable binaries compiled without canaries	-
ELFs	26 executable binaries compiled without canaries and NX	-

bit set, which reinforces the first point made. The third reason is that to take action and -for example- recompile the binaries, a more in depth inspection is necessary to make sure that recompiling the code with NX or SSP enabled will not break any compatibility. This inspection should be done on all the binaries one by one, and it is out of the scope of this project.
The most important binaries to check, especially given the discussion in section 3.6.2, are the ones related to the webserver, Apache2. This machine is used in fact to host two websites through Apache2 and therefore the webserver could represent a vulnerability in the system. TED reports that *apache2* binary is using the SSP, although *rabin2* reports that no canaries are used. Also, few shared objects in Apache2 modules are compiled without SSP/canaries. Both of these observations might require a further investigation, but might be intended and therefore do not require immediate action, although it could be possible to use tools such as AppArmor [4] or SELinux [5] to specify the resources that these binaries can access.
The server in object has very few additional programs installed, and all of them have been installed through the package manager, therefore it is no surprise that not many issues with binaries are found. However, as it has been reported in Table 23, the kernel is vulnerable to a number of exploits which require immediate attention.

Actions. The actions taken on the server are the ones mentioned in Table 23. More in detail, the DCCP kernel module has been disabled (to address CVE20176074), the user namespace

[4] `https://gitlab.com/apparmor/apparmor/wikis/home/`
[5] `https://selinuxproject.org/page/Main_Page`

usage has been restricted to privileged users only (to address CVE20177308) and the kernel has been upgrade to version 4.16.0 (to address CVE20162384, dirtyCOW, dirtyCOW poke variant, Spectre and Meltdown).

Result. After having performed the actions mentioned above, TED has been run again. The new execution of TED took 152m52s and the results are summarized in the Table 24 below.

Table 24. Summary of the results obtained with TED after the hardening of the server.

Type	Desription	Resolved
System	Kernel vulnerable to dirtyCOW	X
System	Kernel vulnerable to dirtyCOW poke variant	X
System	Kernel vulnerable to CVE20177308[1]	X
System	Kernel vulnerable to CVE20162384[2]	X
System	Kernel vulnerable to CVE20176074[3]	X
System	Machine vulnerable to Spectre variant 1	X
System	Machine vulnerable to Spectre variant 2	X
System	Machine vulnerable to Meltdown	X

It is clear from the results obtained that all the critical issues that TED presented in the first report, and for which an action was established, are not applicable anymore. It is possible to conclude that TED allowed to effectively detect vulnerabilities in the system and also it facilitated the creation of an action plan composed of informed choices, and therefore the test can be considered successful.

6.3.2 Locked Shields 2018 infrastructure test

The author has received the authorization to run TED on a portion of the infrastructure used by blue teams during the NATO Locked Shields 2018 cyber exercise [4]. This test consists in running TED on two sets of machines: the first set includes undefended machines, meaning that on them no action has been performed during the exercise, whereas the other set is composed by two machines which belong and have been defended by the winning team. For each team, the machines used for testing are webservers which run their applications inside Docker containers. Such servers represent a very relevant use case for TED, since they are publicly reachable (and therefore could potentially be exploited through a vulnerability -for example- in the webserver software) and they already contain Docker, slimming down the dependencies that TED requires. It is worth mentioning that because Docker by default

[4]`https://ccdcoe.org/largest-international-live-fire-cyber-defence-exercise-world-be-launched-next-week.html`

stores the filesystem of all the containers inside the */var/lib/Docker* directory, TED is able to scan also the binaries present inside the containers.

Offline machines. Although it has been mentioned in a previous section that TED is not currently designed to run in an offline machine, all the four servers used for this test did not have any Internet connection (after the game finished) and were accessible by the author only through SSH (Secure Shell) once connected to the game-network. This condition presented the opportunity to verify whether an Internet connection represents a hard dependency for TED. The conclusion reached is that TED can run in offline machines, and as it has been mentioned in Section 6.1, it is possible to simply export and load the necessary Docker images and the required Python packages and their dependencies. It is clear then that in case a machine is completely offline, TED can still be run by packing all the above mentioned resources on a disk or USB drive, without the need for any network connection whatsoever.

Observations. The execution times for TED's test are summarized in Table 25.

Table 25. Summary of execution times for TED in each of the four Locked Shields machines analyzed.

Machine 1 undefended	196m 47s
Machine 1 defended	145m 18s
Machine 2 undefended	89m 41s
Machine 2 defended	133m 7s

Comparing the results of TED's scans on the first server of each set (Machine 1 undefended and Machine 1 defended) it is possible to observe that the blue team performed some updates (*sshd* and *apache2* binaries changed for example) and of course installed new software (*ossec*, *splunk* etc.) but no major modifications to binaries in order to reinforce their defenses have been performed. One exception is represented by the *mysql* binary that scored 40 points in the undefended machine, but only 20 in the defended one, where canaries have been added, although this could be the result of an update as well. The most interesting observation is that in both cases the system score reported by TED is the same, in particular, the same kernel vulnerabilities are present in all the machines, among which the most interesting is the dirtyCOW. This exploit grants a privilege escalation and is especially relevant in contexts where applications run inside Docker containers, since it can allow an attacker to break out of such containers [1]. The same observations can be done for the second pair of machines, where the exact same differences can be observed.

[1] `https://github.com/scumjr/dirtycow-vdso`

A summary of the most interesting findings is reported in Table 26.

Table 26. Summary of TED's findings on the four Locked Shields machines analyzed.

	Machine 1 undef	**Machine 1 def**	**Machine 2 undef**	**Machine 2 def**
dirtyCOW vulnerable	X	X	X	X
packet ring vulnerable	X	X	X	X
ASLR level	2	2	2	2
apache binaries	-	hash changed	-	hash changed
mysql binaries	-	hardened	-	hardened
ssh binaries	-	hash changed	-	hash changed
postgresql binaries	-	-	-	unchanged
nginx binaries	-	-	-	hash changed

Considerations and results. There are few elements to keep into consideration while outlining the results of this test. The first is that one key score during the execution of Locked Shields exercise is the availability. Many points are lost if a machine is not available, and therefore blue teams tend to avoid rebooting the servers. The second observation is that it is likely that the vector to attack these machines was based on web applications, rather than on binary exploitation, and therefore this is a plausible reason for which not much effort has been put in hardening binary defenses, especially taking in consideration the really short time blue teams have to set their defenses in place.
With the above considerations in mind, the results of the test can be summed up in the following points:

1. TED has very limited dependencies and can run offline.
2. TED is able to scan also the binaries inside Docker containers, if containers are run in Docker with default settings.
3. More tools and defense techniques might need to be integrated by TED to give more detailed information. In fact, it is not possible to completely understand what are the changes applied to binaries. One possibility is to add a system that can -if possible- verify the version of the binary, or compare it with the official package distribution.
4. The complete overview given by TED about the binaries allows to easily track down the known interesting binaries, but makes it hard and time demanding to find out new and maybe malicious or unwanted executables.

5. TED provides a convenient overview of the system defenses. In particular, the integration of Kernelpop allows to quickly check if the kernel is vulnerable to known exploits. In this particular case, it might have been a conscious choice for the blue team not to patch or upgrade the kernel (in order not to lose availability score during a reboot), but TED would have identified such vulnerabilities quickly and therefore it could have contributed to harden the system further.

6. Given the points 3 and 4, TED could fit in a context such as the Locked Shields exercise especially if the defending team knows on which binaries to focus (for example, webserver binaries). In such case, TED's execution time could be reduced drastically and also the report would have a much narrower focus allowing the defenders to quickly assess the risk on meaningful binaries, to individuate possible system vulnerabilities and to correlate the findings.

7. This test allowed to extend the future work for this project.

7 Summary, conclusions and future work

The number of the attacks exploiting binaries vulnerabilities shows that these are relevant as much as they are powerful. Combining this information with the fact that Linux is the dominating server operating system, this project focused on designing and providing a tool that can assist administrators or investigators in assessing the risk in regards to binaries and their execution environment; the tool produced, TED, takes advantage of new technologies, in particular of containers, with the purpose of delivering a product which is portable, scalable, efficient and in line with the current technological trends. In order to develop TED, the most common attacks against binaries have been analyzed, in order to build a realistic and relevant threat landscape, and with these attacks in mind, a thorough analysis and review of the existing defense techniques and mechanisms have been performed. Such analysis showed that despite the very big amount of research done to provide state-of-the-art techniques against the most sophisticated attacks, the number of these techniques that are publicly available and commonly used is really small. In fact, the industry standards in terms of binary protection are still tools relatively outdated. One possible example to stress this point is that despite researchers showed that the default, GCC integrated, Stack Smashing Protector has some weaknesses and despite the fact that hardened versions have been developed, the common binaries found on regular operating systems still employ the standard version as a protection. The same applies when dealing with protections from heap-based attacks and the situation is even worse when dealing with reverse engineering protections; as of today, at the best of the author's knowledge, there is no standard tool to protect an ELF file from reverse engineering. The implementation of TED relied heavily on the analysis just described, and therefore reflected the observation made about the discrepancy between academic research and industry in the choice of the defense measures included. TED in fact verifies that standard mechanisms, from ASLR to NX and from Stack Smashing Protector to stripping, are used, recognizing that new, more powerful tools could make the ones employed obsolete, but also that these will not likely be found in common server deployments and therefore would hinder the usefulness and applicability of TED.
A very successful result has been obtained by the use of containers. The technology chosen, Docker, is the current industry standard and is rich of functionalities. Employing containers allowed an easy integration of multiple tools and a drastic reduction on the dependencies needed, facilitating incredibly the deployment of TED. The tests performed showed that the use of this technology entail a minimal performance loss, making the choice of containerizing the application very beneficial.
Finally, the outcome of this thesis, TED, showed good results in its original purpose, allowing

to have an effective overview of the system in object and facilitating the detection of misconfiguration of possible vulnerabilities. The author believes, after developing this project, that containers are a very suitable technology for building security oriented tools, and that given the satisfactory results produced, TED represents a project that can realistically be used in real-world scenarios and extended in the future.

7.1 Future work

The project developed opened several scenarios that can lead to further research or additional functionalities. In order to improve TED, a large number of modifications can be made, but among them it is worth mentioning the most interesting in the author's opinion. Despite TED runs all the checks it performs inside containers, the main application still runs on the native machine. Fully containerizing the application, including the supporting functionalities, will reduce the dependencies needed to the sole Docker platform, making the tool extremely suitable for cloud environments, where containers are already used in many cases and where the Docker platform often is preinstalled. In addition, this would allow TED do be deployed easily with orchestration tools such as Kubernetes, increasing drastically the scope and the target audience. In terms of functionalities, TED can be easily adapted to scan hosts remotely. The Docker API works over HTTP, therefore one of the main future functionalities that can be added includes the use of remote Docker API to schedule the test containers on a remote host and simply collecting the logs from them, making it no different from a local execution. Concerning security tests and defense verifications, one interesting functionality that can be implemented is the check of functions used in a binary. This means that an ELF file could be examined to verify that inside it no unsafe functions are used. Such verification requires additional research and consideration of several cases of linking and compiling, but it would help identifying unsafe binary, emulating in a sense a very simple static code analysis. Moreover, the checks made to verify hardware-based vulnerabilities could be extended by including a test for the vulnerability to Rowhammer attack. Another important addition that can be made to TED's functionalities is to integrate a tool such as ClamAV [1] to verify at runtime not only whether the binary is protected, but also if it is malicious. Finally, one field in which more research is needed is the protection of binaries from reverse engineering. Except for obfuscation techniques, very little has been done in this field, and therefore both the automated verification of code obfuscation and the development of alternative strategies to protect ELFs from reverse engineering represent valid topics for a PhD research.

[1]https://www.clamav.net/

References

[1] Acbuild official reference. https://github.com/containers/build. Accessed on 11/02/2018.

[2] Burneye official reference. http://www.woodmann.com/collaborative/tools/index.php/Burneye. Accessed on 30/01/2018.

[3] Cgroups man page. http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed on 10th February 2018.

[4] Coreos official reference. https://coreos.com/os/docs/latest/. Accessed 10/02/2018.

[5] *Docker documentation*. https://docs.docker.com/install/linux/linux-postinstall/#manage-docker-as-a-non-root-user. Accessed on 10/02/2018.

[6] Docker python package official reference. https://github.com/docker/docker-py. Accessed on 11/04/2018.

[7] *elf(5) Linux User's Manual*, 2018.

[8] Lxc official reference. https://linuxcontainers.org/lxc/introduction/. Accessed on 10/02/2018.

[9] Lxd official reference. https://linuxcontainers.org/lxd/introduction/. Accessed on 10/02/2018.

[10] Malloc internals. https://sourceware.org/glibc/wiki/MallocInternals. Accessed on 17/09/2017.

[11] National vulnerability database. https://nvd.nist.gov. Accessed on 11/01/2018.

[12] Rancheros official reference. https://rancher.com/rancher-os/. Accessed on 10/02/2018.

[13] Rkt distributions official reference. https://coreos.com/rkt/docs/latest/distributions.html. Accessed 11/02/2018.

[14] Rkt logging official reference. https://coreos.com/rkt/docs/latest/commands.html#logging. Accessed on 11/02/2018.

[15] Rkt official reference. `https://coreos.com/rkt/docs/latest/`. Accessed on 11/02/2018.

[16] *Strip Linux User's manual page*. `https://sourceware.org/binutils/docs/binutils/strip.html`. Accessed on 30/1/2018.

[17] Understanding the stack. `https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html`. Accessed on 17/09/2017.

[18] Unlink macro fix commit. `https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=3e030bd5f9fa57f79a509565b5de6a1c0360d953`. Accessed on 13/12/2017.

[19] wget buffer overflows in processing http data lets remote users execute arbitrary code. `https://securitytracker.com/id/1039661`. Accessed on 11/04/2018.

[20] *The 8086 User's manual*. INTEL corporation, 1979.

[21] Namespaces in operation, part 1: namespaces overview. `http://lwn.net/Articles/531114/`, 2014. Accessed on 27/08/2017.

[22] Usage of operating systems for websites, 2017. `https://w3techs.com/technologies/overview/operating_system/all`.Accessed on 14/10/2017.

[23] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, 2014. USENIX Association.

[24] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. 08 2001.

[25] Emery Berger and Benjamin Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on programming language design and implementation*, PLDI '06, pages 158–168. ACM, June 2006.

[26] Sandrine Blazy and Stéphanie Riaud. Measuring the robustness of source program obfuscation - studying the impact of compiler optimizations on the obfuscation of c programs. 2014.

[27] Tyler Borland. Modern linux exploit development. Slides of the presentation.

[28] Marwan Burelle. *A Malloc tutorial*. EPITA system and security laboratory, 2009.

[29] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-chung Yew. Control flow obfuscation with information flow tracking. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 391–400, New York, NY, USA, 2009. ACM.

[30] Xiaoquan Chen, Rui Xue, and Chuankun Wu. Timely address space rerandomization for resisting code reuse attacks. *Concurrency and Computation*, 29(16), August 2017.

[31] Young-Hyun Choi, Min-Woo Park, Jung-Ho Eom, and Tai-Myoung Chung. Dynamic binary analyzer for scanning vulnerabilities with taint analysis. *Multimedia Tools and Applications*, 74(7):2301–2320, April 2015.

[32] C. Chongkyung Kil, J. Jinsuk Jim, J. Bookholt, J. Xu, and J. Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. pages 339–348. IEEE, December 2006.

[33] Chris Coakley, Jay Freeman, and Robert Dick. Next-generation protection against reverse engineering. 01 2005.

[34] TIS Committee. *Tool Interface Standard. Executable Linkable Format Specification*, 1.2 edition, May 1995.

[35] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, May 2015.

[36] DistroWatch.com. Most popular linux server distributions. `https://distrowatch.com/search.php?ostype=Linux&category=Server&origin=All&basedon=All&notbasedon=None&desktop=All&architecture=All&package=All&rolling=All&isosize=All&netinstall=All&language=All&defaultinit=All&status=Active#simple`. Accessed on 20/04/2018.

[37] Tom Doeppner. x64 cheat sheet. 2016. `https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf`. Accessed on 1/10/2017.

[38] Chris Eagle. Strike/counter-strike: Reverse engineering shiva. `https://www.blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf`. Accessed on 30/1/2018.

[39] Eldad Eilam. *Reversing. Secrets of Reverse Engineering*. Wiley Publishing, 2005.

[40] Ryan "elfmaster" O'Neill. *Learning Linux Binary Analysis*. Packt, 2016.

[41] Daniel Regalado et al. *Gray Hat Hacking*. McGrawHill, 2015.

[42] Tang Feng-Yi, Feng Chao, and Tang Chao-Jing. Memory vulnerability diagnosis for binary program. *ITM Web of Conferences*, 7, January 2016.

[43] Agner Fog. *Calling Conventions for different C++ compilers and operating systems*. Technical University of Denmark, 05 2017.

[44] Anley C. Heasman F. Lindner F. Richarte G. *The Shellcoder's handbook. Discovering and exploiting security holes*. Wiley Publishing, 2007.

[45] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. *IEEE Security Privacy*, 5(3):82–84, May 2007.

[46] GB_MASTER. X86 exploitation 101: Heap overflows, August 2014. `https://gbmaster.wordpress.com/2014/08/11/x86-exploitation-101-heap-overflows-unlink-me-would-you-please/`. Accessed on 1/10/2017.

[47] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: Long live kaslr. volume 10379, pages 161–176. Springer Verlag, 2017.

[48] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[49] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? pages 571–585. IEEE, May 2012.

[50] Fu-Hau Hsu, Cheng-Hsien Huang, Chi-Hsien Hsu, Chih-Wen Ou, Li-Han Chen, and Ping-Cheng Chiu. Hsp: A solution against heap sprays. *The Journal of Systems & Software*, 83(11):2227–2236, 2010.

[51] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, May 2013.

[52] Intel. *System V Application Binary Interface - AMD64 Architecture Processor Supplement*, 2016.

[53] Kaspersky Security Intelligence. Ics vulnerabillities 2015. Technical report, Kaspersky Lab, 2015. `http://newsroom.kaspersky.eu/fileadmin/user_upload/de/Downloads/PDFs/ICS_Report_Part1_Vulnerabilities.pdf`.

[54] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[55] Georgia Killcrece, Klaus Peter Kossakowski, Robin Ruefle, and Mark Zajicek. State of the practice of computer security incident response teams (csirts). Technical report, 2003.

[56] M. J. Kim, J. Y. Lee, H. Y. Chang, S. Cho, Y. Park, M. Park, and P. A. Wilsey. Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 80–86, May 2010.

[57] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. January 2018.

[58] Benjamin Kuperman, Carla Brodley, Hilmi Ozdoganoglu, T Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, November 2005.

[59] McAfee Labs. Threats report. Technical report, Intel Security, 2017.

[60] Zhiqiang Lin, Bing Mao, and Li Xie. Libsafexp: A practical and transparent tool for run-time buffer overflow preventions. In *2006 IEEE Information Assurance Workshop*, pages 332–339, June 2006.

[61] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. January 2018.

[62] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 280–291, New York, NY, USA, 2015. ACM.

[63] Raymond Lutui. A multidisciplinary digital forensic investigation process model. *Business Horizons*, 59(6):593–604, 2016.

[64] H. Marco-Gisbert and I. Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In *2013 IEEE 12th International Symposium on Network Computing and Applications*, pages 243–250, Aug 2013.

[65] Microsoft. Cve-2017-11779 | windows dnsapi remote code execution vulnerability. Technical report, Microsoft, 2017. `https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2017-11779`. Accessed on 14/10/2017.

[66] Adrian Mouat. *Using Docker*. O'Reilly, 2016.

[67] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. July 2013.

[68] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[69] Association of Chief Police Officers. Acpo good practice guide. Technical report, Police Central e-Crime Unit, 2012.

[70] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[71] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. pages 601–615. IEEE, May 2012.

[72] Crispin Cowan Perry Wagle. Stackguard: Simple stack smash protection for gcc. 2003. `ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf`. Accessed on 7/12/2017.

[73] Richard Johnson Peter Silberman. A comparison of buffer overflow prevention implementations and weaknesses. iDEFENSE inc., 2004.

[74] Robin Ruefle, Audrey Dorofee, David Mundie, Allen D. Householder, Michael Murray, and Samuel J. Perl. Computer security incident response team development and evolution. *Security & Privacy, IEEE*, 12(5):16–26, September 2014.

[75] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. pages 298–307, 2004.

[76] Neel Mehta Shaun Clowes. Shiva. advances in elf binary encryption. 2003.

[77] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013.

[78] Jaydeep Solanki, Aenik Shah, and Manik Lal Das. Secure patrol: Patrolling against buffer overflow exploits. *Information Security Journal: A Global Perspective*, pages 1–11, November 2014.

[79] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In R. Sekar and Arun K. Pujari, editors, *Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[80] PaX Team. mprotect. `https://web.archive.org/web/20160809013618/http://pax.grsecurity.net/docs/mprotect.txt`. Accessed on 10/12/2017.

[81] PaX Team. Non-executable pages design and implementations. `https://web.archive.org/web/20160809013618/http://pax.grsecurity.net/docs/noexec.txt`. Accessed on 1/12/2017.

[82] PaX Team. Address space layout randomization. 2003. `https://web.archive.org/web/20160809013618/http://pax.grsecurity.net/docs/aslr.txt` Accessed on 1/12/2017.

[83] PaX Team. Overall pax description. 2003. `https://web.archive.org/web/20160809013618/http://pax.grsecurity.net/docs/pax.txt`. Accessed on 1/12/2017.

[84] Frederick Ulrich. *Exploitability Assessment with TEASER*. Msc thesis, Northeastern University, 2017.

[85] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, Frank Piessens, Somesh Jha, and Anish Mathuria. Valueguard: Protection of native applications against data-only buffer overflows. December 2010.

[86] w3techs.com. Usage statistics and market share of linux for websites. Technical report, 2018. `https://w3techs.com/technologies/details/os-linux/all/all`.Accessed on 07/02/2018.

[87] Run Wang, Pei Liu, Lei Zhao, Yueqiang Cheng, and Lina Wang. deexploit: Identifying misuses of input data to diagnose memory-corruption exploits at the binary level. *The Journal of Systems & Software*, 124:153–168, February 2017.

[88] Zhi Wang, Renquan Cheng, and Debin Gao. *Revisiting Address Space Randomization*, pages 207–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[89] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. volume 4307, pages 379–398. Springer Verlag, 2006.

[90] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438, Dec 2006.

[91] Yves Younan. 25 years of vulnerabillities:1988-2012. Technical report, Sourcefire Vulnerability Research Team, 2012.

[92] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. *SIGPLAN Not.*, 46(6):367–377, June 2011.

Appendix 1 - ASLR verification supporting binary

Listing 1. Custom binary used to support ASLR validation.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void print_env_address(){
    extern char **environ;
    printf("ENV␣=␣%p\n",&environ[0]);
}

void print_stack_pointer() {
  void* p = NULL;
  printf("STACK␣=␣%p\n", (void*)&p);
}

void print_heap_address(){
  int* p=malloc(1);
  printf("HEAP␣=␣%p\n", p);
}

void main() {
    print_env_address();
    print_stack_pointer();
    print_heap_address();
    exit(EXIT_SUCCESS);
}
```

Appendix 2 - Tampering script used for functional test

Listing 2. Custom script written to introduce vulnerable binaries in the system.

```
#!/bin/sh
#First, download coreutils source code
cd /tmp
wget http://ftp.gnu.org/gnu/coreutils/coreutils-5.0.tar.gz
tar -xvzf coreutils-5.0.tar.gz
cd coreutils-5.0/
#Compile them with unsecure flags
./configure CFLAGS='-fno-stack-protector -z execstack'
make
cd src/
#Move all ELFs compiled in /bin directory (overwriting original ones)
for file in $(ls .); do
        file $file | grep ELF
        if [  $? -eq 0 ]
            then
                echo moving $file
            mv $file /bin/
        fi
    done
#Second, download random insecure C program
wget https://raw.githubusercontent.com/npapernot/buffer-overflow-attack\
     /master/stack.c -O /tmp/insecure.c
#Third, disable ASLR
sysctl kernel.randomize_va_space=$(( RANDOM % 2))
#Pick random binary in /sbin
file=$(ls /sbin | sort -R | head -1)
echo $file
#Finally, compile  without ssp and add to /sbin the insecure binary
cd /tmp
gcc -o insecure -fno-stack-protector insecure.c
mv /tmp/insecure /sbin/$file
#Cleanup
rm /tmp/insecure.c /tmp/coreutils* -r
```

Appendix 3 - Example of (redacted) TED report

Listing 3. Example of TED report, redacted removing most of the binaries.

```
{
    "ELFs": {
        "/bin/bash": {
            "ELF score": "0/100",
            "NoExec": {
                "Score": 0,
                "Stack_executable": false,
                "W^X enforced": true,
                "nx_flag": true
            },
            "Stack_Smashing_Protector": {
                "Score": 0,
                "canaries": true,
                "stack_chk_fail": true
            },
            "Stripped": {
                "Binary stripped": true,
                "Score": 0
            },
            "sha256": "3d74ea46393deee0ff74bfbfd9479242
                      b090f20ebffe9a27f572679f277153ed"
        },
        "/bin/bunzip2": {
            "ELF score": "0/100",
            "NoExec": {
                "Score": 0,
                "Stack_executable": false,
                "W^X enforced": true,
                "nx_flag": true
            },
            "Stack_Smashing_Protector": {
                "Score": 0,
                "canaries": true,
            "stack_chk_fail": true
        },
        "Stripped": {
            "Binary stripped": true,
            "Score": 0
        },
        "sha256": "4924a82483afcb65207d8185053f4387
                   050cd3a96e9c1a58c6d86e32400c6be4"
    },
    "/bin/busybox": {
        "ELF score": "40/100",
        "NoExec": {
            "Score": 0,
            "Stack_executable": false,
            "W^X enforced": true,
            "nx_flag": true
        },
        "Stack_Smashing_Protector": {
            "Score": 40,
            "canaries": false,
            "stack_chk_fail": false
        },
        "Stripped": {
            "Binary stripped": true,
            "Score": 0
        },
        "sha256": "b6b1d844400bd319cac244aa51329fb8
                   4f8a19cb64fb0fa364f6f88ea866bde2"
    },
    "/bin/bzcat": {
        "ELF score": "0/100",
        "NoExec": {
            "Score": 0,
            "Stack_executable": false,
            "W^X enforced": true,
            "nx_flag": true
            },
            "Stack_Smashing_Protector": {
                "Score": 0,
                "canaries": true,
                "stack_chk_fail": true
            },
            "Stripped": {
                "Binary stripped": true,
                "Score": 0
            },
            "sha256": "4924a82483afcb65207d8185053f4387
                      050cd3a96e9c1a58c6d86e32400c6be4"
        },
        "/bin/bzip2": {
            "ELF score": "0/100",
            "NoExec": {
                "Score": 0,
                "Stack_executable": false,
                "W^X enforced": true,
                "nx_flag": true
            },
            "Stack_Smashing_Protector": {
                "Score": 0,
                "canaries": true,
                "stack_chk_fail": true
            },
            "Stripped": {
                "Binary stripped": true,
                "Score": 0
            },
            "sha256": "4924a82483afcb65207d8185053f4387
                      050cd3a96e9c1a58c6d86e32400c6be4"
        },
[Redacted 152983 lines]
        "/var/lib/dpkg/info/bash.preinst": {
            "ELF score": "10/100",
        "NoExec": {
            "Score": 0,
            "Stack_executable": false,
            "W^X enforced": true,
            "nx_flag": true
        },
        "Stack_Smashing_Protector": {
            "Score": 0,
            "canaries": true,
            "stack_chk_fail": true
        },
        "Stripped": {
            "Binary stripped": false,
            "Score": 10
        },
        "sha256": "f214b9b229a16f050f7c268a22c786b8
                   da70196f5996b5872065994bdb4e3bbc"
    },
    "Description": "A set of checks is made on each ELF
        file. The check includes three main areas: 1)
        NoExec, which consists of three independent tests
        to verify that the sections of each ELF that do
        not contain code are NOT flagged as executable.
        The first test checks if the stack is marked as
        executable.The second test checks that there are
        no sections in the whole ELF which are marked both
         Writable and eXecutable (W^X). The third test
        checks that rabin2 output on the elf includes the
        nx flag. rabin2 checks that the GNU_STACK section
        is not executable, similarly to what is done in
        the first test. 2)Stripped. This test simply
        verifies if the ELF has been stripped. 3)Stack
        smashing protector. This test is composed of two
        subtests:The first test checks the output of
        rabin2 and verifies that the canaries flag is
        present.The second test uses objdump to
        disassemble the ELF and verifies thatthe code
        contains a call to stack_chk_fail."
},
"System checks": {
    "ASLR": {
        "ASLR_hard_check": "ASLR is enabled and applied
           also for data segment.",
        "ASLR_hard_value": "2",
        "ASLR_soft_check": "ASLR is enabled and applied
           also for data segment.",
        "ASLR_soft_value": "2",
        "Description": "This check verifies that ASLR is
           in place and also to what extent. The check
           itself is made in two ways, here called soft
           and hard. The soft check queries the kernel
           parameter randomize_va_space via the sysctl
           command. The hard check instead runs a simple
           binary that prints out the address of the
           stack pointer, the address of the system
           environmental variables and the address
           returned by a malloc call. This operation is
           done 10 times and then it is verified whether
           there are duplicates in the addresses returned
           .",
        "Score": 5
    },
    "Kernelpop": {
        "Description": "This check uses kernelpop tool to
            determine whether the running kernel is
           vulnerable to known exploits. In order to do
           so, a database with exploit is maintained. For
            more info check https://github.com/
           spencerdodd/kernelpop",
        "Score": 27.375,
        "confirmed": [
            {
```

```
            "cve": "CVE20165195_64_poke",
            "description": "Dirty COW race condition
               root priv esc for 64 bit (poke variant
               )",
            "reliability": " HIGH"
        },
        {
            "cve": "CVE20177308",
            "description": "`packet_set_ring` in net/
               packet/af_packet.c can gain privileges
                via crafted system calls.",
            "reliability": " HIGH"
        },
        {
            "cve": "CVE20165195_64",
            "description": "Dirty COW race condition
               root priv esc for 64 bit",
            "reliability": " HIGH"
        },
        {
            "cve": "CVE20162384",
            "description": "Double free vulnerability
                in the `snd_usbmidi_create` (requires
                physical proximity)",
            "reliability": "LOW"
        },
        {
            "cve": "CVE20176074",
            "description": "`dccp_rcv_state_process`
               in net/dccp/input.c mishandles structs
                and can lead to local root",
            "reliability": "LOW"
        }
    ],
    "potential": [
        "No potential exploits"
        ]
    },
    "Nx_support": {
        "Description": "This simple check verifies only
           that the CPU supports the nx bit.If this is
           true, it means that all the executable should
           also flag the memory segments that do not
           contain code as non-executable. All modern (64
           bit) CPUs should support the nx-bit, which
           allows the Operating systems to flag memory
           pages as Non-Exec by setting this bit to 1.
           For 32bit machines, this bit is available only
            if processor supports PAE.",
        "Score": 0,
        "nx_supported": true
    },
    "Spectre_meltdown": {
        "Description": "This test uses a third party
           script to check if the system isvulnerable to
           three variants of the Spectre and Meltdown
           attacks.The script can be found at https://
           github.com/speed47/spectre-meltdown-checker",
        "Score": 20,
        "Variant 1": {
            "action": "Kernel source needs to be patched
               to mitigate the vulnerability",
            "cve": "CVE-2017-5753",
            "vulnerable": true
        },
        "Variant 2": {
            "action": "IBRS hardware + kernel support OR
               kernel with retpoline are needed to
               mitigate the vulnerability",
            "cve": "CVE-2017-5715",
            "vulnerable": true
        },
            "Variant 3": {
                "action": "PTI is needed to mitigate the
                   vulnerability",
                "cve": "CVE-2017-5754",
                "vulnerable": true
            }
        },
        "System score": "47.35/100"
    },
    "time": "2018-04-09 21:40:39.087527",
    "type": "full"
}
```