POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

PRIVMUL:

PRIVilege separation for Multi-User Logic applications

Relatore: Prof. Federico Maggi

Correlatore: Prof. William Robertson, Northeastern University

Prof. Stefano Zanero

Tesi di Laurea Specialistica di:

Andrea Mambretti

Matricola n.783286

Anno Accademico 2013–2014

“Life is a lot like Jazz...it’s best when you improvise.”

George Gershwin

Contents

List of Figures

List of Listings

List of Abbreviations

C.G. Call Graph. 37, 62

CPU Central Processing Unit. vii, 27, 52

G.V. Global Variable. 4, 24, 29, 36, 37

IPC Inter-Process Communication. 3, 13, 17, 18

LKM Linux Kernel Modules. 11–13, 50

O.S. Operating System. xii, 2, 4–6, 11–13, 17, 20, 23–25, 30–34, 43, 44, 46, 47, 60, 61, 63–66

Sommario

In ogni sistema informatico, bug e vulnerabilità sono da sempre causa di problemi di sicurezza e di malfunzionamenti. Con la complessità sempre crescente del *software* aumenta in proporzione anche il numero di problemi riscontrati. Tra i principali effetti collaterali che bug e vulnerabilità possono portare la *privilege escalation* ha un ruolo sicuramente primario, poichè permette ad un aggressore di sfruttare delle vulnerabilità per eseguire operazioni afferenti a livelli di privilegio più elevati di quelli dell'utente di base, ad esempio.

Molte soluzioni sono state proposte sia per eradicare il problema sia per identificare e bloccare gli attacchi, ovvero i tentativi di sfruttare le suddette vulnerabilità. Tutti i lavori presentati si concentrano sul problema di privilege escalation di un attaccante a livello di sistema operativo, ossia l'evitare che un aggressore riesca, sfruttando vulnerabilità a programmi privilegiati, nell'acquisizione di diritti di amministrazione che gli permetterebbero di prendere il controllo della macchina. È stato inoltre dimostrato che non solo i sistemi operativi sono affetti da questo problema ma anche applicazioni con una logica multiutente, cioè applicazioni che mostrano, in funzione della tipologia di utente, solo una sotto parte delle funzionalità di tutta l'applicazione.

Il mio lavoro propone un meccanismo di *privilege separation* contro questi tipi di attacco a supporto di applicazioni con logica multiutente. Il mio sistema, PRIVMUL, propone un nuovo modo per scrivere o adattare questo tipo di applicazioni e ne garantisce un'esecuzione sicura, con lo scopo di proteggere funzionalità e dati appartenenti a profili diversi dal profilo che sta eseguendo l'applicazione. PRIVMUL integra applicazioni, scritte utilizzando le API messe a disposizione, con il sistema operativo che ne garantisce un'esecuzione sicura.

PRIVMUL annulla i privilegi di accesso delle pagine di memoria ai profili diversi da quello che sta eseguendo l'applicazione. Per fare ciò ho creato un meccanismo che permette allo

sviluppatore di specificare dei profili utente al momento della compilazione dell'applicazione. PRIVMUL inoltre si propone di proteggere da possibili leak di dati sempre a causa di vulnerabilità presenti nell'applicazione. Per eliminare questa possibilità PRIVMUL traccia ogni dato creato e distrutto all'interno dell'applicazione ne impedisce l'accesso non autorizzato. PRIVMUL, per essere flessibile, supporta il cambio di profilo durante l'esecuzione attraverso un meccanismo di autenticazione.

I risultati sperimentali indicano che l'aumento del tempo di esecuzione introdotto dal meccanismo di autenticazione di PRIVMUL non compromette l'usabilità dell'applicazione. I test di tracciamento delle allocazioni presentano invece un aumento consistente del tempo di esecuzione se il numero di allocazioni dinamiche è elevato. Ad ogni modo, considerando il tipo di applicazione per cui PRIVMUL è stato disegnato, e cioè applicazioni con una grossa interazione con l'utente, l'aumento del tempo di esecuzione rimane trascurabile se comparato con il tempo operativo dell'utente.

Summary

In every computer system, bugs and vulnerabilities have always been cause of many security problems and malfunctioning. The size and the number of people that design and build these systems are growing and with them also the probability that mistakes are accidentally inserted. *Privilege escalation* is one of the most well known problems correlated with bugs and vulnerabilities where an attacker, exploiting a privileged software, is able to execute functionalities that belong to privilege levels higher than his/her user profile.

Several works in this area try to either prevent or defeat privilege escalation in computer systems. Most of them work on the specific privilege separation inside the Operating System (O.S.), trying to avoid that the exploitation of a privileged software brings an attacker to gain administrative privilege level that will allow the attacker to have the machine control. However, a recent work shows that not only operating systems are suitable for privilege escalation but also multi-user logic application could suffer from this issue. It shows how a user could access functionalities of other users' profiles exploiting the new GEM vulnerabilities class.

In this work, I present a new security mechanism that applies *privilege separation* to multi-user logic application to defeat privilege escalation and data leak. PRIVMUL, the system I present, provides a new mechanism to write or adapt this kind of application. PRIVMUL links the application to the operating system, through new provided APIs, that guarantee the safe execution of every operation.

The approach I propose protects the code and data of other profiles from unauthorized access by the current profile that is running the application. To achieve this protection PRIVMUL temporarly removes all the access rights on the memory pages to profiles that are different from the current one. It also aims to protect dynamic data providing a chunk based tracker along with the support of the operating system. PRIVMUL, to be flexible, supports

a runtime profile switch through an authentication mechanism.

The experimental results show that this approach introduces an acceptable overhead that does not impact the overall usability of the protected application.

Chapter 1

Introduction

In modern software systems, due to their complexity, it is not hard to find security bugs, also known as *vulnerabilities*, which can be leveraged by malicious adversaries to manipulate the control or data flow of a program during its execution.

For example, I want to highlight the vulnerability CVE-2009-0065 that was found in the Linux kernel (v 2.6) where the attacker was able to exploit a bug inside the SCTP stack in a reliable way. This bug exploit allowed to provide a remote connect-back shell on all x86-64 hosts running that specific kernel [31]. Another example of vulnerability is the set composed by CVE-2013-0977, CVE-2013-0987 and CVE-2013-0981 where the attacker was able to privilege escalate the iOS operating system avoiding the system sandbox and breaking out the virtual machine guest. It was counted that this set of three vulnerabilities was working on at least 5 million of iPhone devices during February 2013 [32]. Only these two cited cases by themselves could target million of machines and devices all over the Internet. They could target private users' devices as well as production servers making users' and companies' softwares and data at risk. Looking at the Secunia report [23] for the number of vulnerabilities found in the past few years, I can see a growing trend, from 8369 vulnerabilities in 2008 to 13073 in 2013.

The latter case of vulnerability falls in the category of privilege escalation bugs, because it allows an attacker to execute code or access data he or she was not meant to. It shows what I can achieve thanks to privilege escalation and data leaking through memory corruption vulnerabilities. The source of the problem is rooted in the intrinsic difficulty of designing and implementing a correct access control mechanism, a problem which becomes very challenging

in complex software. In particular, software designers and programmers need to create proper privilege distribution management and enforcement procedures and make sure that, from compile to runtime, no errors are introduced. This is clearly very challenging if the language or programming environment offer no support for, or when it is simply too expensive to redesign an existing large system. The bigger the systems are the greater the number of security problems brought by mistakes in their management is. So the management of privileges among the components of a system has always been a huge problem to solve. In general, the privilege distribution problem could be almost completely solved during the system design phase, by defining which are the least privileges each component needs or which isolation I can apply to the component itself [22]. However, in modern software industry what usually happens is that either the design and development costs or the performance constraints lead to a monolithic approach of constructing the system. Without a deep analysis on how the system could be divided into several independent and cooperative pieces, reducing the risk of improper use of them is limited.

For an attacker who takes over a monolithic system, it is simpler to control it, if the privileges are the same for every component. To explain this idea with a concrete example I can think of an O.S. where all the components have root privileges (executed by root). In this case if an attacker is able to gain the control (that means having arbitrary code execution) over of one them, such as the network component, he/she can then increase his/her control doing everything (e.g., writing in every location he/she wants inside the filesystem). With proper *Privilege Separation* or *Isolation* this would not be either so easy or possible at all.

Depending on the system I am considering (generic application, operating system etc.) the possible solutions change. There are solutions that try to prevent this kind of issue and others that try to detect and block the execution when an improper privilege escalation is detected. Some of the detection approaches try to verify that either data or the execution flow or both are correct when the application is running. Some of those techniques are described in Chapter 3. Instead, among the preventing approaches there are approaches that define access policies such as SELinux [24] to restrict privileges on a per-application base and others that define new ways to build this sensitive applications, for instance, creating different processes that cooperate instead of having a single monolithic piece of code. The work presented in this thesis is contained by the latter kind of approach. Related to this I should first mention

Privilege Separation and *Least Privilege*, which have been firstly introduced by Provos et al. in [21], that are two possible approaches which can be used to limit or to avoid security problems.

The *Privilege Separation* is the practice of dividing an application/system into two parts, a privileged one and an unprivileged one. This is done to reduce the privileged part to the minimum dimension possible, in such a way I can reduce the probability that inside that part of the program there are vulnerabilities. The smaller the privileged component is, the lower the risk of vulnerabilities is. Usually the implementation of this concept represents two parts as two distinct processes that communicate through Inter-Process Communication (IPC).

The *Least Privilege* is the practice of creating multiple components out of a single application/system, each component being created by a single significant operation. The difficult aspect here is to find the least number of instructions to accomplish the operation. Every component is represented by a process. The main difference between *Privilege Separation* and *Least Privilege* is that the second one is more fine grain than the first one. *Least Privilege* does not represent macro functionalities but it tries to protect basic operations.

In the last few years, significant progress has been made to apply *Privilege Separation* and *Least Privilege* concepts to already existing programs. [21] shows how *Privilege Separation* can be applied to an already existing program manually. Clearly, the main disadvantage of this solution is that manually dividing a big application implies long operations and that who is dividing the application should know the whole code base to avoid to break the divided program flow somehow. In [14], the author wants to improve the separation process by providing a library that includes all the basic operations needed in the partitioning phase. The main drawback of this solution is that it still requires manual operations to divide an application. Moreover, if I want to privilege separate an operation that is not included in the provided set I must patch the library to add it that could mean the introduction of new security flaw. [3] is the first work that tries to separate the privileges of the program in a semi-automatic way.

Regarding the concept of *Least Privilege*, the work [33] by Wu et al. proposes to use the dynamic data dependency analysis on many traces of execution of an application to define different components. This approach really depends on how good the execution traces are so if the coverage does not include all the possible scenarios the whole partitioning system does not work properly.

In this work, however, I slightly modify and extend the *Privilege Separation* approach,

considering several parts kept in the same process. First, I consider multiple privilege levels; this means that I can have a whole hierarchy of privileges and not only the two standard levels proposed by Provos (the admin and the normal level). Second, I consider applications with multi-user logics. This is important because I can apply my approach to all of those programs where I can define more than two different users, like a company application where different users from different area of the company use the same program to access different functionalities. I can protect one user profile's functions from improper access by the users of other area of the company. None of the previous approaches has ever considered to protect this particular kind of applications that are critical almost as well as all the server daemons that are the goals of all the other works. Third, I achieve the previous two points while keeping original application architecture unaltered. This makes my system less obtrusive in terms of migration costs. Moreover, since my solution does not require to split the original application architecture into multiple processes running at different privilege levels, no IPC overhead is introduced.

In this thesis with PRIVMUL, I introduce an innovative approach to design, write and run multi-user logic applications. Under the hood, PRIVMUL creates a strong and flexible link between the application and the O.S. that guarantees, in a non compromised kernel, fully secure execution also in case of vulnerable applications due to developers' mistakes.

My key observation is that privilege separation, in application with multi-user logic, allows an attacker to execute code that does not belong to his/her profile and this happens because the code of the other profiles is loaded along with the code of the attacker profile. To this end the key intuition is to *protect* all the code and data of other profiles as far as the active profile is not eligible to access them. In this way even if the application has multiple profiles, the user, for the whole session, can run only the functionalities of his/her profile. The same idea of protecting perfectly solves the data leak problem. In this case the kind of data with which I should work are Global Variable (G.V.) and dynamic allocated data.

In my system, I use the same mechanism of tagging explained by Brumley, but the attribute I define permits to express an infinite number of levels instead of the only two (priv/unpriv) defined in his work. This tagging mechanism allows to specify the different privilege levels inside the source code of the application. PRIVMUL takes the annotated source code and builds the binary keeping in mind every privilege level defined. It writes into the binary meta-

data, describing the privilege levels layout, that are read by the O.S. when the application is launched. O.S. guarantees the **protection** of higher privilege levels when a lower privilege level is running. It also validates transition among privilege levels providing an authentication mechanism when the application wants to access an higher privilege level. During the application execution the O.S. keeps track of different memory allocations and **protect** them from other privilege levels as well as it is done for the application functionalities.

In summary, my contributions are the following:

- I propose PRIVMUL, a novel code-tagging mechanism in LLVM/Clang to design applications with multi-user logic while avoiding privilege escalation or user data leaking due to memory corruption vulnerabilities.
- I implemented PRIVMUL as a GNU/Linux kernel module that supports application with $N > 2$ different levels of privilege inside the application logic.
- I designed and implemented a mechanism to track and protect dynamically allocated data during execution.

Chapter 2

Background

Any software may contain vulnerabilities. In most cases, vulnerabilities could bring serious consequences including the compromise of the overall security of a system. A motivated attacker could do various things if he or she is able to find and exploit those vulnerabilities. The most common kinds of attacks allow to steal information in a company network, to compromise and remotely control a machine to do illegal operations, to escalate privileges inside a system, to cause denial of service and so forth.

In Section 2.1 and 2.2 I introduce the background concepts on software vulnerabilities. I focus on the classes of vulnerabilities that lead to privilege escalation or data leak in multi-user logic applications. Those include both classic (e.g., memory corruption) and recent vulnerabilities (e.g., GUI Element Misuse [18]). The generality of the PRIVMUL approach makes it applicable to prevent vulnerability classes that are not described here for brevity. PRIVMUL is based upon LLVM, Linux, ELF and Glibc. In 2.2.1 I introduce LLVM/Clang compiler infrastructure, giving a brief description of its structure with a focus on the parts used in PRIVMUL. In 2.2.2 I introduce the Linux kernel, the O.S. I used for the implementation. In 2.2.3, I describe the ELF format, focusing on its header because PRIVMUL uses it to store all the metadata attached to the application executable. In Section 2.2.4, I present the Glibc libraries, with particular emphasis on the loader and the memory allocator.

2.1 Vulnerabilities

2.1.1 Memory Corruption Vulnerabilities

Memory Corruption Attacks refers to attacks that allow an attacker to deterministically alter the execution flow of a program by submitting crafted input to an application [29]. In general, a necessary condition for memory corruption vulnerabilities to occur is that the program language demands the memory management to the programmers. The two most famous languages where I can find this class of vulnerabilities are C and C++, which are still used when performance is paramount or when the hardware should be programmed directly.

Table 2.1: Programming Language Statistics

Programming Language	**Nov 2013**	**Nov 2014**	**Ratings**
C	1	1	17.469%
Java	2	2	14.391%
Objective-C	3	3	9.063%
C++	4	4	9.063 %
C#	5	5	4.985%

Programming language diffusion statistics from TIOBE software

Looking at some statistics, it is clear that C is still the most used language along with Java, and C++ is in the Top5 [27].

Memory corruption vulnerabilities have been gaining increasing attention and are still present in modern software. It is worth mentioning some of the milestones that have marked the history of these vulnerabilities. The first documented attack that exploited a memory corruption vulnerability has been found in 1972 and is described in a US Air Force document [1].

By supplying addresses outside the space allocated to the users programs, it is often possible to get the monitor to obtain anauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash.

There exist several ways to exploit memory errors. Depending on the specific nature of the vulnerability, an attacker reads-writes some primitives to implement exploits, and corrupts

different memory areas. For a thorough historical overview, I refer the reader to [29], and limit my attention to the basis of the main techniques. One of the first techniques still common today is the stack-based buffer overflow presented in 1996 by Elias Levy (also known as Aleph One) in [19]. This attack was already known many years before Levy's work but he was the first who published an detailed systematization. In fact in 1988, Robert Tappan Morris Jr. wrote and released the *Morris Worm*, the first computer worm distributed on the Internet, which propagated through the exploitation of a buffer overflow inside the fingerd daemon [29]. In this technique, an attacker is able to get arbitrary code execution using a developer's mistake in the management of a buffer index. For example considering the following code:

Listing 2.1: Vulnerable Program to Buffer Overflow Example

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

int main(void)
{
    char buff[15];
    int password = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "p4ssw0rd"))
    {
        printf ("\n Wrong Password \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        password = 1;
    }

    if(password)
    {
       /* Now Give root or admin rights to user*/
        printf ("\n Root privileges gained \n");
    }

    return 0;
}
```

In the Listing 2.1, the program asks a user the password to gain privileges. Below, it is shown a possible interaction that exploits the overflow to gain privileges without actually inserting the correct password.

Listing 2.2: Exploitation Example

```
>./overflow

 Enter the password :
AAAAAAAAAAAAAAAAAAAA

 Wrong Password

 Root privileges gained
```

An attacker can easily exploit the application above. He/She has only to insert more characters than the buffer length (e.g., Listing 2.2). The developer's mistake is that the input length is not controlled and the attacker has full control on the content of the buffer on the stack. This implies an overflow of the buffer and all the cells of memory contiguously to the buffer are overwritten as well. In this example the next variable after the buffer is the *password* variable that is used to evaluate the authentication and it is initialized to zero. After the input of the string, the *password* variable is overwritten by the overflow and its value is not zero anymore. Once the execution reaches the password check, the program recognizes that the password is invalid but the program keeps allowing the attacker to access to the privileged function.

Even such a simple vulnerability could allow an attacker to do privilege escalation if the bug is located in a privileged piece of code. Another famous type of memory corruption is the format string bug, first discovered in 1999 by Tymm Twillman during security auditing the proftpd daemon [28]. The problem was known before that date but [28] is the first correctly crafted attack that could be used to privilege escalate inside a system. This bug is introduced when an attacker is able to control the format string (e.g., printf). Supplying placeholders such as %x, %n, %u in a proper order permits to access and overwrite memory on the stack including for instance the saved instruction pointer register values of the saved function frame.

Furthermore, the list of memory corruption vulnerabilities includes heap-based vulnerabilities that can allow code execution and possibly privilege escalation. Two of these attacks are the "double free", explained in [10], and the heap overflow, explained in [5], [26] and [17]. One of the first heap-based attacks was crafted in 1998 by Christien Rioux for Windows 95.

The attacks listed above are only main examples, variations of them could be found every day to improve the reliability on the attack success or avoid some new security mechanism.

2.1.2 GUI Element Misuse Vulnerabilities

GUI element misuse is a new class of vulnerability discovered by Mulliner et al. [18]. GUIs are very common in nowadays applications because they are more intuitive for basic users than the command line. It is very easy to control the operations of a system using the mouse or visual menus instead of typing long commands. Essentially, the authors found many applications where the privilege levels policies were enforced by the use of the GUI features rather than the usual policy schemes the operating system was offering. They discovered cases where the application was using a disabled button to deny the access to a certain functionality instead of using the system-level policy functions.

This kind of vulnerability is not based on the execution of other pieces of code provided by an attacker but it allows an attacker to access functionalities that should not be accessible to him/her. With this class of vulnerability all the memory protections implemented so far are totally ineffective because they allow execution of code that is already present in the vulnerable program. Notably, PRIVMUL covers this class of vulnerability because it enforces application policies at system rather than at application level (or, worse, at GUI level). System policies can be tampered with by an attacker only if the attacker has root access to the machine.

2.2 Toolchain

2.2.1 LLVM/Clang Compiler Infrastructure

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [16]. It started as a research project and now is one of the most developed compilers. Its popularity is due to its flexibility and extensibility. It is composed by several other projects. The most important are the *Clang* front-end and the *LLVM Core* libraries.

Clang is a C/C++/Objective C and Objective C++ front-end for LLVM. It has a very good expressive diagnostics to identify bugs and problems in a program and it supports the C++11 standard syntax. It is a fast front-end with a low memory use compared to the main competitors such as GCC. It supports also GCC syntax elements (e.g., specific attributes). In Section 5 I describe how I modified the front and back end to allow the programmer to specify access-control policies by means of new attributes for global variable and function.

The *LLVM Core* libraries provide a modern source- and target-independent optimizer. This compiler back-end receives the program from the front-end already transformed in a intermediate representation. It has two different chains of optimization passes. The first chain that elaborates the code works on the intermediate representation and it brings general and common transformations to the code. After that, the transformed intermediate representation code is translated into machine code of the specified target. The produced machine code is further elaborated to optimize the code for the specific target. A common optimization is the if-conversion, which works on architectures that support predicated instructions (e.g., ARM and Hexagon).

2.2.2 The Linux Kernel

"Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single Unix Specification compliance."[30]

Probably, Linux is one of the most well known open source Unix-like O.S.. It has always been under heavy development since its first release back in 1991. It is for sure the most targeted O.S. for research purposes. Its applicability goes from the desktop computers to servers passing through embedded devices (such as routers, tablets, smartphones) and super computers. Furthermore, it supports several architectures, the main being x86, x86_64, ARM, Sparc and MIPS.

The Linux kernel offers many development interfaces. For the purpose of this work I provide a brief summary of Kernel Modules (2.2.2), System Calls (2.2.2), Netlink Interfaces (2.2.2).

Kernel Modules

Linux Kernel Modules (LKM) are the most used ways to extend the Linux kernel. In many configurations they do not require a re-compilation of the whole kernel but only of the module source code files. In fact, compiled modules can be loaded and unloaded at runtime without influencing the overall execution of the O.S..

LKM are mainly used for [15]:

- **Device Drivers** - All the drivers that allow to control an external piece of hardware such as a webcam.
- **Filesystem Drivers** - LKM allow to extend the Linux kernel to support non standard filesystem. Two main examples of this usage are the famous FUSE project [12] and the ZFS support [34].
- **System Calls** - Instead of including system call in the kernel image it is possible, using LKM, to add it without modifying the main kernel image. Usually this is the suggested way to include new system calls because the Linux kernel developers try to keep the system call table as stable as possible.
- **Network Drivers** - LKM could be used to include support either for a new or for a custom network protocol.
- **TTY line disciplines** - With LKM it is possible to write terminal drivers too. For example you can create your own TTY for the console that automatically corrects what you are typing or you can write your own serial port to emulate a serial communication.
- **Executable Interpreters** - An executable interpreter loads and runs an executable. Linux is designed to be able to run executable in various formats, and each one must have its own executable interpreter. For instance, the Linux kernel does not natively support the Java bytecode. Using a LKM I can add this feature to the kernel and run Java bytecode as a standard executable.

System calls

System calls represent the most basic and used interface to the kernel functionalities. They are mainly used to request the kernel special operations that may require either the use of hardware components or privileged operations. In the Linux kernel system calls are identified by a number. Of course, they change according to the supported architectures. This means that, for instance, the read system call has a different identifier in x86, in x86_64 and in ARM. In Linux the system call interfaces are pretty stable and they are changed very infrequently by the kernel developers. For this reason system call should be avoided if you want to integrate some functionalities in the kernel mainline. In other O.S. such as Microsoft Windows system

calls change in every version, in fact Microsoft suggests to use wrapper libraries to guarantee compatibility across versions.

Netlink Sockets

Netlink sockets, as described in [13], are a special IPC to send information back and forth between kernel and user-space processes. This kind of communication is preferred to system calls, ioctls or proc filesystem because it is very easily integrable with the kernel and, from the user-space application point of view, it is like using a traditional socket. Netlink sockets are pretty flexible because every netlink could have its own protocol. Netlink is asynchronous because it provides a socket queue to smooth the burst of messages. From the kernel perspective only the protocol number should be added to the netlink.h file and a LKM should be written to configure the interface for the user-space applications.

2.2.3 ELF (Executable and Linkable Format)

Since 1999, the ELF is the standard binary format for executable under all the Unix O.S. [9]. It has been developed by the Unix System Laboratory for the System V release 4 (SVR4) Application Binary Interface (ABI).

The ELF format is flexible and extensible. It represents both object files and executable files. In the object representation (Linking View) the file is divided into *Sections*. In the executable representation (Execution View) the file is divided and loaded into *Segments*.

The ELF format is composed by several different headers. The main one is the ELF header that contains all the information related to all the other headers (section and segment headers) such as the offset from the beginning of the executable binary (*e_phoff* and *e_shoff*), the number of segments (*e_phoff*) or sections (*e_shoff*) that are inside the file. Its structure is depicted in the Listing 2.3.

Listing 2.3: Elf Header Structure

```
typedef struct {
    unsigned char   e_ident[EI_NIDENT];
    Elf32_Half      e_type;
    Elf32_Half      e_machine;
    Elf32_Word      e_version;
    Elf32_Addr      e_entry;
```

```
    Elf32_Off       e_phoff;
    Elf32_Off       e_shoff;
    Elf32_Word      e_flags;
    Elf32_Half      e_ehsize;
    Elf32_Half      e_phentsize;
    Elf32_Half      e_phnum;
    Elf32_Half      e_shentsize;
    Elf32_Half      e_shnum;
    Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

Every section is described by one header. Sections contain the information in an object file, except the ELF header, the program header table, and the section header table. The C struct that represents a section header is depicted in the Listing 2.4.

Listing 2.4: Section Header Structure

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

The most notable fields of this struct are *sh_name*, which is the index into the section header string table, *sh_offset*, which represents the offset from the beginning of the file to the first byte in the section, and *sh_size*, which represents the size in bytes. In my system, as described in Chapter 5, I use these parameters to save the information for every privilege level (user profile).

The ELF format has several standard sections such as *.text*, where the code is usually contained, or *.debug* where the debug information, if any, is stored. It also allows developers to introduce new sections at compile time depending on the needs. In my case I used this feature of the ELF format so as to forward useful information, such as the names for every profile, in the process and to give a first profiles partition.

Last, the program header table is an array of structures, each describing a segment. In a segment, one or more sections could be contained. The C struct that represents one segment header is the Listing 2.5.

Listing 2.5: Segment Header Structure

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;
```

In this struct there are many notable fields; one of this is the *p_type*. This field content changes the meaning of the other fields in the struct. Possible values are *PT_LOAD* that means loadable segment (*.text*, *.bss*, *.data* and other are contained into this kind of segments that are mapped by the dynamic loader in memory), *PT_PHDR* that specifies the location and the size of the program header table itself, and *PT_NULL* for the empty or meaningless segments that are ignored.

2.2.4 Glibc

Glibc is a C library version from the GNU project. It is used as main C library in most of the systems with Linux kernel. it provides interfaces to all the common system calls of the Linux kernel to allow application compatibility among different versions of the kernel. For the purpose of this work, I need to highlight the loader and the allocator.

Loader

The Glibc loader is called ld.so. In the execution chain, its job is to open a binary, read the ELF and the segments headers and map all the *PT_LOAD* segments into memory. Then, it starts reading all the shared libraries used by the application and tries to load and map them as well. In Section 5.3.1 I describe how I modified this element of the Glibc to achieve new functionalities needed by my system.

Allocator

The allocator refers to the set of C functions that allow a program to manage the heap. Without this set of functions for a programmer the management of the heap would be still

possible but with a significant effort. The allocator caches pieces of memory when previous allocation are freed by the application and keeps them in a list of chunks. When a new allocation happens, the allocator tries to satisfy the request looking for the cached elements in the list. If those elements cover the request, the allocator removes those chunks from the list and returns the pointer to them, thus saving an expensive call to the glsOS. This operation guarantees a performance improvement for the application.

The Glibc allocator is the standard one and it is for general-purpose applications. It was written by Doug Lea and adapted to multiple threading by Wolfram Gloger. Citing from the source code comment

> This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs.

Other allocators exist and are used for more specialized kinds of applications. They may give better performance for some kind of applications and worse for others.

Chapter 3

State of the Art and Research Gaps

In this chapter I describe the state of the art regarding *Privilege Separation*. The *Privilege Separation* concept was elaborated by Provos et al. in [21] together with the *Least Privilege* concept. On the application of the *Privilege Separation* concept two other main works exist which try to automatize the system application process. The first one is *Privman* proposed by Kilpatrick in [14] and the second one is *Privtrans* proposed by Brumley in [3]. I describe these works and illustrate where and what this thesis contributes.

3.1 Privsep: Preventing Privilege Escalation

In [21] Provos et al. present the concept of *Privilege Separation* and they show how it could be applied to all the privileged services in a Unix O.S.. The operation of *Privilege Separation* is manually done on the source code of the application. In their work the authors take OpenSSH as main example, being OpenSSH one of the most targeted software for remote privilege escalation attacks.

The idea behind this work is to reduce the amount of privileged code as much as possible to lower the probability of programmer's mistakes and so vulnerabilities. They logically divide the application into two different components. The first component is called monitor and it contains all the privileged operations the application may ask. The second component is the unprivileged program and for every privileged operation this second process should ask the monitor to perform it. The monitor and the unprivileged programs are two different processes isolated one from the other. The two processes are allowed to communicate through IPC (such

as shared memory, pipes, sockets).

In the authors' implementation, OpenSSH is divided into two processes and a socket is created to allow the communication between the two elements. The monitor waits for requests from the unprivileged process that manages the connection with the remote user. If the unprivileged process asks for a non permitted operation the monitor terminates. The authors also show that the privilege separated OpenSSH does not penalize performance.

This work achieves *Privilege Separation* through a manual operation on the source code. Moreover, the authors' approach divides the program into two different processes creating an IPC communication. This could be done with small services but the complexity of this operation grows along with the program size. In my approach the *Privilege Separation* is achieved without dividing the binary into multi processes. This makes the application of the *Privilege Separation* easier even with huge programs. Furthermore my approach improves flexibility because it allows multiple levels of privileges. There is no IPC communication among processes but only communication through the kernel to authenticate the more privilege request.

3.2 Privman: A Library for Partitioning Applications

Privman improves the applicability of *Privilege Separation*. The author finds the manual approach to divide application presented in [21] limiting, so he decides to propose a library to guide the process. The author's library allows an application to be divided into a privileged server and a main application. As in [21], the main application has to ask to the server to perform the privileged application. The library already supports most of the usual privileged operations such as file-access functions, PAM authentication, bind() and daemon(). Furthermore, the server could be configured with a series of policies written into a configuration file.

Compared with the approach presented in this thesis, Kilpatrick's library has the same limitations of Privsep. It considers only two privilege levels. Also, it provides only few privilege separation functions. For unsupported functions a developer should include them extending the library itself. In my approach, this issue is not present because the tagging mechanism works independently from the underline function that should be privilege separated.

3.3 Privtrans: Automatically Partitioning Programs for Privilege Separation

Privtrans is one of the most relevant works on *Privilege Separation*. Brumley et al. propose an approach where the *Privilege Separation* is applied in an automatic way. The authors provide the developers a new mechanism of annotation that allows to define, in a monolithic software, which parts are privileged and which ones are not. The mechanism of tagging is provided through a C attribute with one parameter. The parameter of the attribute could be either **priv** or **unpriv** depending on the case. Privtrans receives the annotated source code and applies a C to C transformation. At the end it produces two distinct programs, one called *monitor* where there are all the privileged operations and a *slave* where there are the unprivileged operations.

Moreover the authors extend the applicability of the *Privilege Separation* concept considering also a distributed setting where the *monitor* is not running on the same machine of the *slave*. When the code partition happens, Privtrans replaces all the privileged calls with wrappers that supports RPC. Using this mechanism they can hide the monitor location thanks to the RPC library they are providing, and place it either locally or remotely without impact the modification they are bringing to the slave code.

As introduced by Privman, also here the monitor supports the specification of policies that permit or deny operations. These policies are written in C code inside the monitor itself. Also in this work the authors use OpenSSH to evaluate their implementation.

Privtrans has some common characteristic with the work presented in this thesis. Also in my work there is an annotation mechanism to identify the privilege levels inside a certain application but in my work the number of possible levels is variable and not fixed to **priv** and **unpriv**. However, my tagging mechanism does not aim to split the application. Instead, it aims to keep the monolithic application structure (very hard to be split into multiple processes) and just to internally reorder the program itself. If the new RPC library is vulnerable some privileged functionalities may be exposed to the slave without the correct authorization allowing privilege escalation. They also demand the creation of certain new privileged operation to developers that could introduce mistakes in that very delicate interface. The main difference between my approach and Brumley's approach is that Privtrans mitigates

the privilege escalation problem through privilege separation in user-space, while, PRIVMUL enforces policies through the O.S. that makes the vulnerable surface much smaller and hard to exploit.

3.4 Related Work

In this section I overview the main techniques used today to mitigate memory corruption vulnerabilities and give some comments on why they do not solve the problem entirely. Since the first memory corruption vulnerability was found, several research works have been proposed to mitigated this class of vulnerabilities, by make the attacks either unreliable or not possible at all. The most relevant protection mechanisms are ASLR, StackGuard, StackShield, StackGhost, Non-Executable Data Pages and FormatGuard.

3.4.1 Address Space Layout Randomization

ASLR makes many attacks unreliable. It maps applications to random address spaces, which change at each execution. With ASLR, the attacker must guess (or brute force) where the elements (shellcode, libraries, code segment) are loaded to successfully exploit the application. The success probability for an exploit decreases along with the amount of feedback an attacker receives from the application. Moreover, If the attack is performed over the network and in the system a back-up process to restart the failed application is not present, the attacker has only one possible chance to exploit the application. This makes the ASLR protection mechanism very effective.

However, against this security mechanism, Tyler Durden in [8] finds a possible leak of the running programs address space thanks to a partial IP overwrite. This shows that the overall idea is pretty strong to defeat various exploits but its implementation is crucial.

3.4.2 StackGuard

This technique, presented by Cowan et al. in [7], works at compile time. It could work in two different ways; detecting the change of the return address before the function returns, or preventing any overwriting of the return address.

The detection of modification is achieved by using a *canary* placed before the saved return

address on the stack. This *canary* is checked before the function returns to the saved address location and if the *canary* is corrupted it means that an attempt of overwriting has happened. This makes the overwrite of the saved IP address very complicated and feasible only in case an attacker could precisely overwrite the address bytes on the stack. The authors in this work also presents an enhancement of the *canary* where they randomly choose the *canary* values. This random *canary* is picked up using the *crt0* library when the program starts. In this way it would be harder for an attacker to craft a reliable overflow string due to the always different *canary* on the stack.

In [7], the authors also present a tool called *MemGuard* designed to protect the return address when a function is called and remove the protection when the function returns. This protection is similar to a *canary*. It is composed by a *prologue* and an *epilogue*, which are responsible for the protection and un-protection of the saved IP on the stack. This protection mechanism introduces an overhead of 1800 times for a write operation, so the authors suggest to use it only for debbuging.

However [4] demonstrates how to exploit stack overflow vulnerabilities even if StackGuard is used. Even more, the authors of this attack consider an environment where the stack is non-executable. The only assumptions are:

- A pointer located on the stack after the buffer
- Overflow bug that allows to overwrite the pointer
- One `*copy()` function (e.g., `strcpy`, `memcopy`) that takes the pointer as destination and user-specified data as the source, and no pointer initialization between the overflow and the `copy`.

3.4.3 StackShield

This attack also works for StackShield that is a GCC extension. Stackshield tries to avoid buffer overflows copying the saved IP at the beginning of the DATA segment where the overflow cannot overwrite. Then, StackShield compares the saved value on the stack and the saved value in the DATA segment and terminates the application if the two values are different.

3.4.4 StackGhost

StackGhost, presented in [11] by Mike Frantzen and Mike Shuey, is an approach to mitigate buffer overflow vulnerabilities and some format string bugs using a Sun Microsystem's Sparc processor architecture feature. The Sparc architecture instead of copying data to and from the stack, it provides a set of registers containing all the function information. This is done to speed-up the process of storing and recovering a previous function frame. The mechanism is similar in spirit to the fastcall calling convention, but here it is used effectively to pass frame information back and forth, and not just to pass parameters.

Every function frame has 24 registers equally divided into input, output and local registers. Inside two contiguous function frames, the eight input registers of the former frame correspond to the eight output registers of the latter frame. The RET address is saved and passed among all the function frames. The problem is that the number of physical registers is limited and I cannot have too many nested function frames. To solve this, the kernel copies the values inside the common registers into the stack, starting from the registers of the oldest frames. This call back to the kernel is where StackGhost plays a key role. In brief, it performs XOR operations between the return pointer and a fixed (or per-process) cookie or by encrypts the saved stack frame.

However, the Sparc architecture already provides strong protection due to the fact that it uses registers instead of the stack if the nested call sequence is not too long. It may happen that StackGhost is never called because the registers are enough for all the function frames.

3.5 Non-executable Data Pages

Many stack-based overflow attacks require that the shellcode and the overflow buffer are both placed on the stack. An attacker usually tries to reach his/her own shellcode on the stack and execute it. By removing the executable flag to the data segments the Grsecurity team (ex PaX team) [25] shows that stack-based buffer overflow exploits are no longer feasible. Many implementations of the "writable XOR executable" concept exist for various architecture. Many architectures allow this specification at hardware level, others such as x86 should implement the concept at the operating system (memory manager) level.

This technique could also be circumvented, as shown by Alexander Peslyak (also known

as Solar Designer) in [20] where he presents a new technique called "return to libc" that cat bypass non-executable data pages protections. In essence, the concept of return to libc consists in executing code that is already present on the victim's machine (i.e., in the libc). This technique has evolved over the years, creating a family of exploitation methods known as return oriented programming (or ROP in short), against which only control-flow-integrity-based protection work.

3.6 FormatGuard

Usually format string happens because the number of % directives is controlled by the attacker and there are not enough elements in the parameter list of the printf to satisfy the % directives number. In this way the printf function starts to read other elements from the stack basing on the % directives it finds that are not supposed to be read. FormatGuard [6] mitigates format-string vulnerabilities. The idea is to count the number of parameters and compare it to the number of % directives found in the format string.

FormatGuard works in the C preprocessor of the compiler, with a set of macros that tries to detect every possible mistake in the format string use. An alert is provided at compile-time if a suspicious case is detected.

3.7 Discussion

The targeted application between my work and the presented works (Privsep, Privman and Privtrans) is different. They consider privilege escalation at O.S. level, instead, mainly I aim multi-user application trying to defeat all kind of vulnerabilities such as the new GEM vulnerability class. None of the previous work has never considered the security of those kind of applications. Furthermore none of them could protect from the GEM attacks. My approach goal is to hide privilege levels while the application is running if the profile is not eligible to access them. So even if the application is vulnerable it is not possible to privilege escalate the system because the privileged operation are not accessible in other memory pages.

Even if my approach has been designed on multi-user application I don't exclude the applicability of my approach to also protect from privilege escalation at O.S. level. This could not be true for all the other works presented in this chapter.

Chapter 4

Proposed Approach

PRIVMUL solves the two distinct problems of privilege escalation and data leaking presented in the previous chapters. To this end, I introduce a new way to design, write and run multi-user logic applications. Under the hood, PRIVMUL creates a strong and flexible link between the application and the O.S. that guarantees, in a non compromised kernel, fully secure execution also in case of vulnerable applications due to developers' mistakes.

My key observation is that privilege escalation, in application with multi-user logic, allows an attacker to execute code that does not belong to his/her profile and this happens because the code of other profiles is loaded along with the code of the attacker profile. Therefore, my key intuition is to *protect* all the code and data of other profiles as far as the active profile is not eligible to access them. In this way even if the application has multiple profiles, each user can run only the functionalities of his/her profile. This also solves the data leaking problem. To this end, I *protect* both G.V. (with a partitioning at compile time) and dynamically allocated data (with a runtime protection).

PRIVMUL is composed by several elements that work in a chain. It starts with the application design and ends with the application execution.

In general, PRIVMUL relies on a tagging mechanism (described in the Section 4.2) that works on the source code of the application. Then, it propagates the tagging (as described in Section 4.3) to achieve effective data and code partitioning (as described in Section 4.4). Once loaded, the application code and data are kept partitioned in their representation in memory, and the O.S. is informed about such partitioning (e.g., name, position in memory of

every privilege level profile). This operation is described in Section 4.5. Only one profile can be loaded and running at the same time and the application can change it through a request to the O.S.. The O.S. is considered the reference monitor that authenticates each change of profile according to the partitioning. This authentication functionality proposed at the O.S. level is described in Section 4.6. O.S. is also designated for tracking all the data that the application uses, keeping record of which profile created those data and allowing only that profile to access it later on. This last operation done by the O.S. is described in Section 4.7.

4.1 Threat and Attacker Model

The case scenario I consider for this work is a large, multi-user logic application that is running on a shared machine. The typical use case is a kiosk application with several profiles, or a large enterprise application (e.g., SAP) installed on a shared machine with several employees each having a distinct profile. Different users can use this machine in different moment in time and each user, depending on his/her own profile, can access some of the functionalities of the application. Our attacker model is a user that has access to the machine and is able to exploit a vulnerability that leads the application to execute functionalities or to read data outside the user (i.e., attacker) authorization profile.

I reduce the trusted element to the kernel, which is the only part that I assume to be non compromised (e.g., no root kit has been planted). Consequently, the attacker has no root-access to the machine, and thus no possibility to tamper the binary file or interfere with the loading phase of the application.

4.2 Tagging mechanism

My approach is based on information inserted in the source code, which can be manually inserted by developers or automatically deducted by the system. Ideally, the **Tagging Mechanism** receives the source code and optionally the tag information and creates the tagged source code. The tag itself could be represented in various way. It could be something added to the code or a particular modification of the code itself (Figure 4.1).

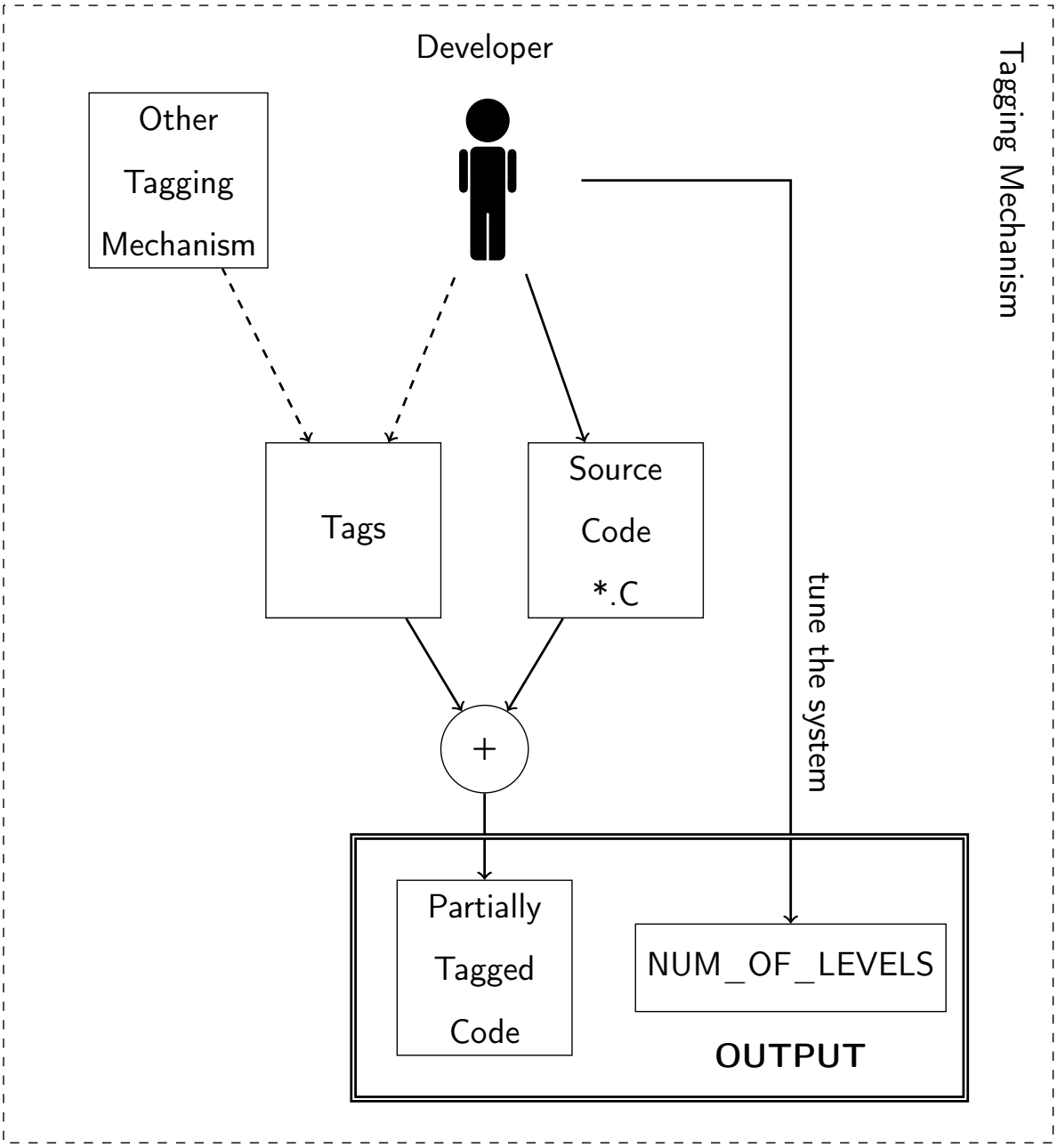

Figure 4.1: This is the first part of the model. The tags could be provided by either developers or some other tagging mechanism. Developers also have to set the global variable called NUM_OF_LEVELS that will be used in all the following automatic operations on the source code and at runtime

The way tagging is performed depends on the decision taken in other modules of the system. I identified two possible schemes. The *onion scheme* (Figure 4.2) where level 3 profiles share code and data with level 2 profile, and so on. In the *tree scheme* (Figure 4.3), admin-level profile (root of the tree) can access every any code and data, whereas leaf-level profile are accessible by any other intermediate level and can access only its code and data. These tagging schemes must result in a source code with the metadata related to single-user profiles included. This might depend on which scheme is elected, the onion or the tree. The output of PRIVMUL contains information related to which piece of code and which global variables belong to a certain user profile.

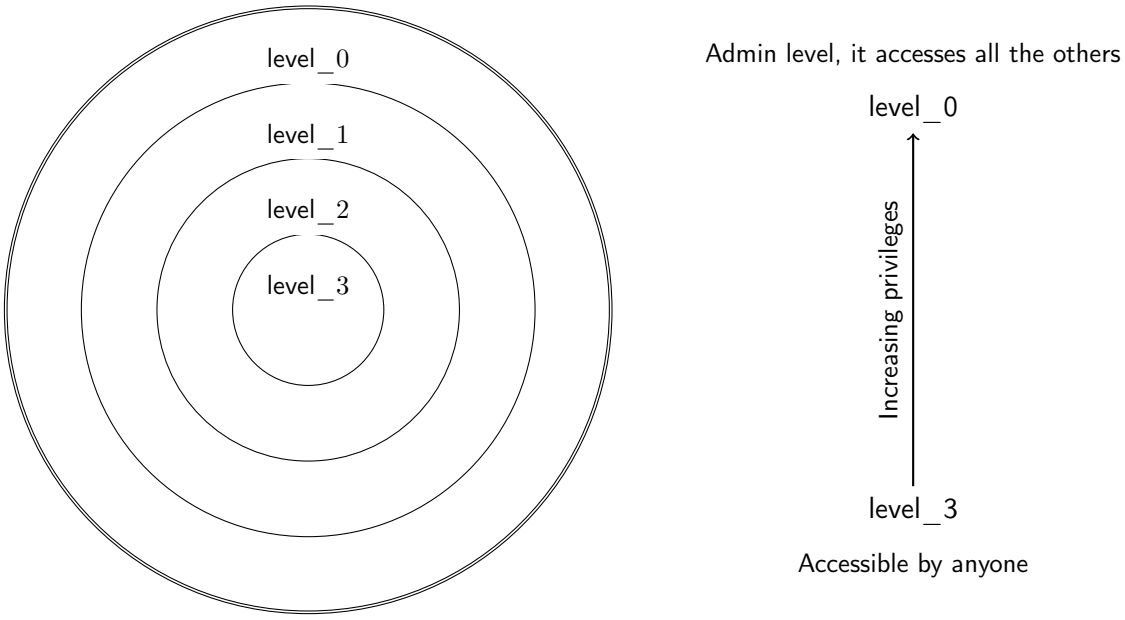

Figure 4.2: Onion scheme of levels. This should not be confused with the CPU ring, it represents other kind of privileges that are application specific. Furthermore in my model the levels go from 0 (administration level) to NUM_OF_LEVEL, represents the most accessible level in the hierarchy. In the example NUM_OF_LEVEL is equal to 3.

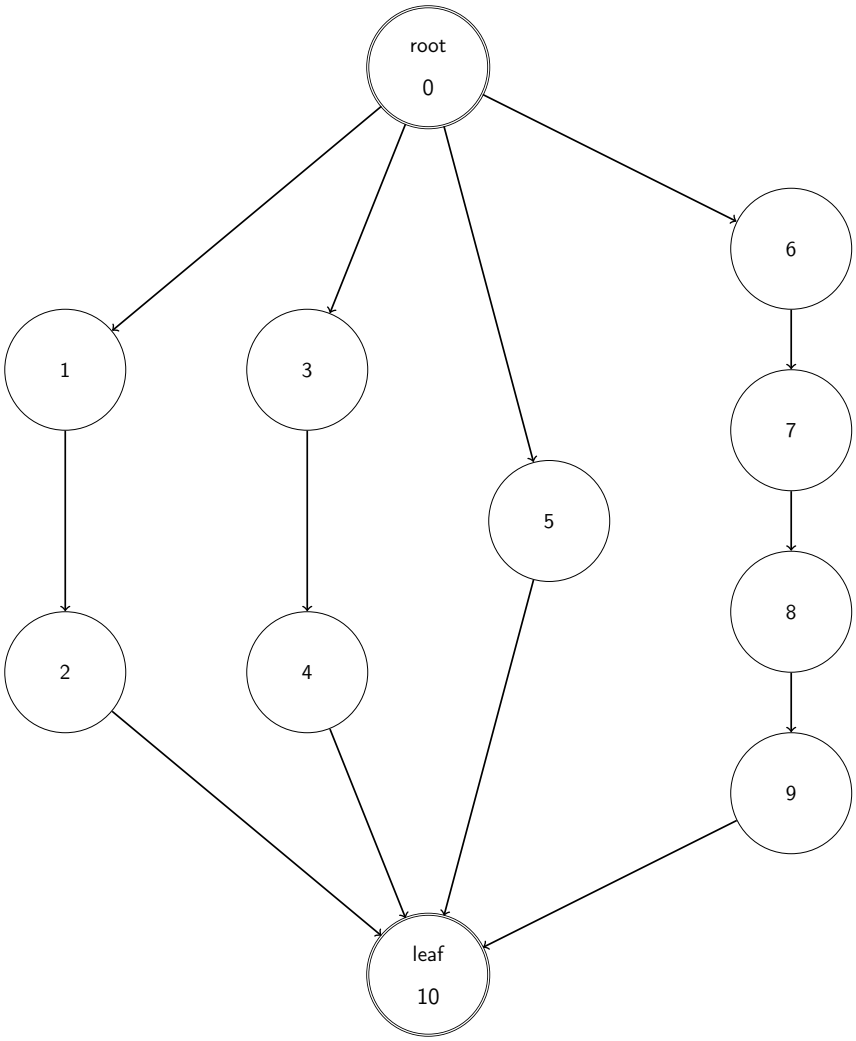

Figure 4.3: Tree scheme of levels. The root of the tree (level 0) represents the admin-level profile that can access every any code and data of all the nodes below. The leaf-level profile (level 10), instead, is accessible by any other intermediate level and can access only its code and data. The intermediate profiles on different branches (such as 4, 5 and 7) do not share neither code nor data among each others.

4.3 Tagging Propagation

The second module of PRIVMUL that analyzes the code is the tagging propagation block (see Figure 4.4), which helps the developers to automatically propagate tagging information. Without this block tagging propagation should be completed manually on all the functions thus increasing the chances that the developer could inadvertently introduce errors. Therefore,

although the task performed by this module is quite simple, it is very important.

In the onion scheme, for each non-tagged callee node in the graph it verifies all the callers and, if all of them have a tag, picks the caller tag most accessible in the onion hierarchy and assigns it to the callee node.

In the tree, If a callee is called by other levels on different branches, to satisfy all the accessibility constraints, it is tagged with the leaf level. For instance considering the level distribution as in Figure 4.3, if I am tagging a non-tagged function called by two other functions from level 3 and 5, the only way this new node could be accessible by both is staying at the bottom of the tree into the leaf-level because the leaf-level is the only level shared by all the branches.

This phase propagates the tags only on the functions. It is not related to the tags of the G.V.. This implies that every G.V. that should not be accessible by somebody must be tagged manually.

During the propagation, the module requests interaction with the developers that can force some decision to the propagation or base all the operations on an algorithm.

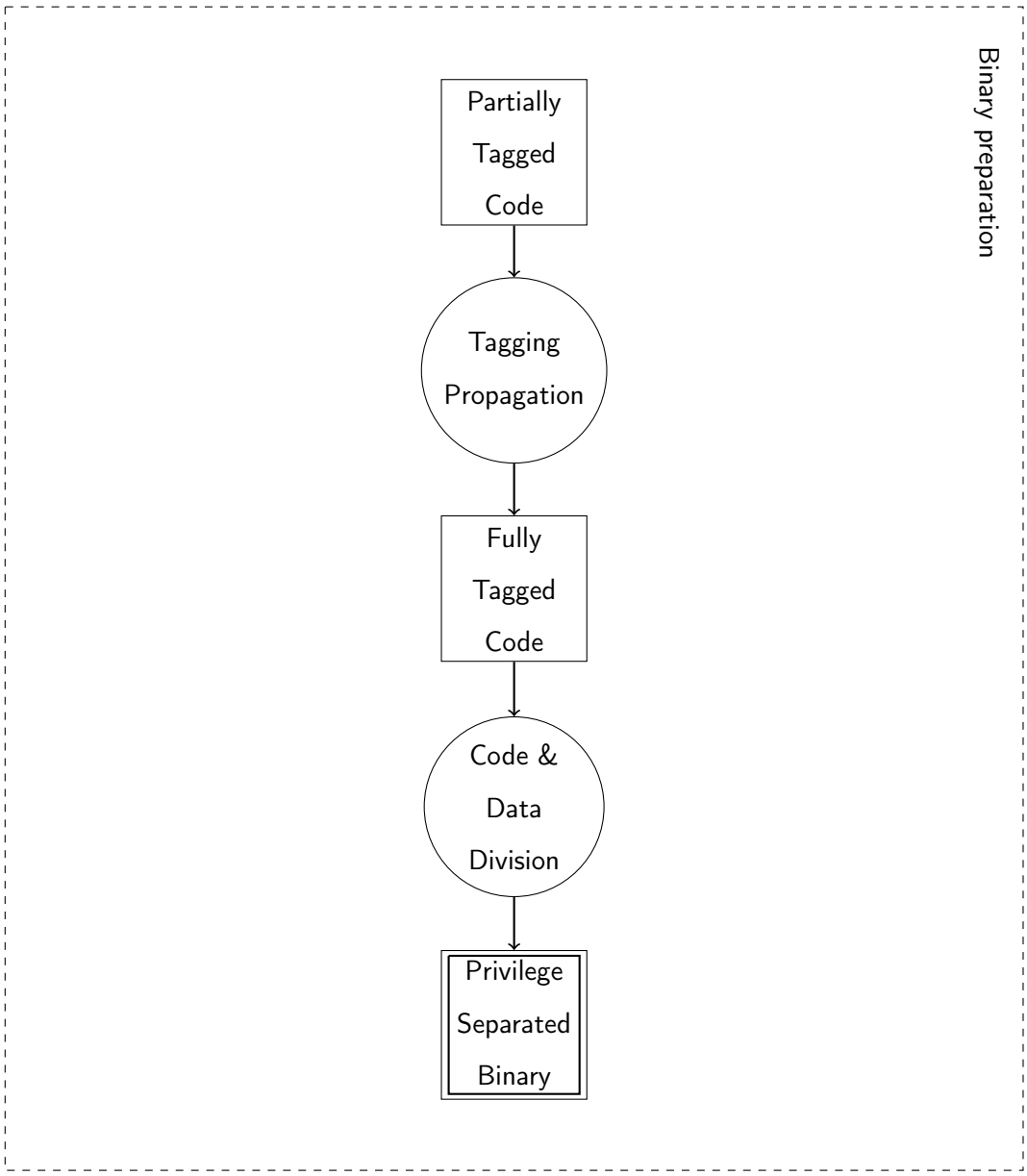

Figure 4.4: This is the second part of the model, where the binary is instrumented for the system through a series of components. The output will be a binary that will contain the metadata required by the privilege-separation support in the loader and O.S. to work.

4.4 Code and Data Separation

In this module (see Figure 4.4) the code with all the propagated tags is processed to be separated. This module is the responsible of the transformation of the tagged source code into an executable binary. The most notable property of the resulting binary is that code and global data are divided into smaller sections; the binary is still a single, standalone file but each section is isolated according to the privilege levels. Together with the binary, this module encodes the metadata about each privilege level. The metadata could be stored in a separated file or inside the binary. The binary layout is very important because the O.S. will be able to protect data and code accesses based on such elements.

4.5 Operating System Initializer

This module of the system initializes the O.S. using the information provided by the previous modules. Therefore, it reads either the binary or the external files produced and then informs the O.S.. The communication between this block and the O.S. occurs before the application execution. The O.S. is informed about the access control model used to correctly manage privilege-level switching.

Referring to Figure 4.5, This module is represented by the connections ① and ②. The ① connection is the first in temporal order and it shows the operation of the binary layout information sending (**Metadata**) to the O.S.. The ② connection, instead, shows the usual dynamic linking operation and the following application execution (**Loading**). After this phase is completed the state of the system is in **Running Program**.

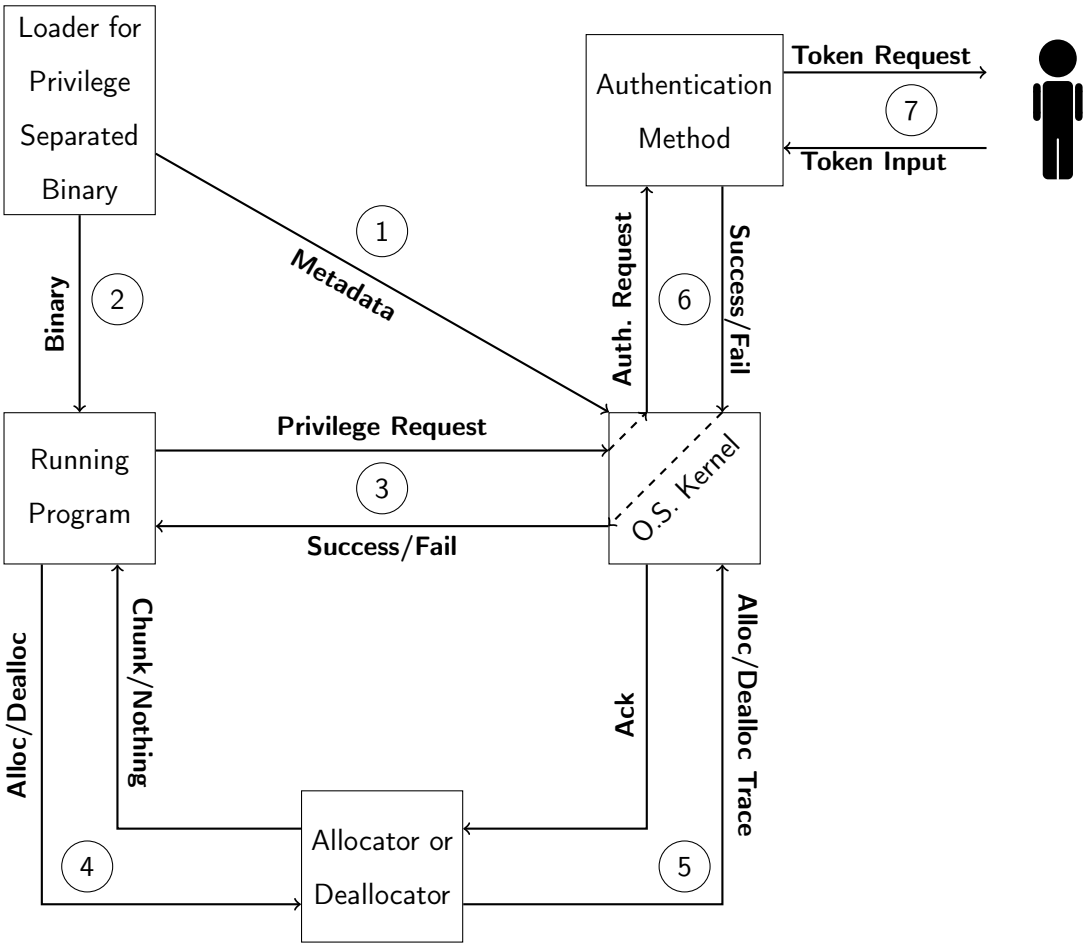

Figure 4.5: Runtime application interaction with PRIVMUL . In this figure it is showed how the application is managed by PRIVMUL from the loading in memory to the end of the execution. With ① and ② are represented the O.S. initialization (**Metadata**) and the **Loading** of the binary. These two operations happen once at the beginning of the execution. After them, the status of the system is in **Running Program**. If a request for switching to a different profile happens (**Privilege Request**) the **Authorization Mechanism** is activated. The **Authorization Mechanism** is composed by the connection ③, ⑥ and ⑦. The program asks to the **O.S. Kernel** to switch. The kernel activates an **Authentication method** (**Auth. Request**), which prompts the user to enter (**Token Request**) a proper authentication token (password, smart card). Depending on the correctness of the token (**Token Input**), a **Success**/**Fail** answer is sent to the **O.S. Kernel**, which will either allow or deny the user-profile switching. Furthermore, the **Running Program** could perform an allocation or deallocation (**Alloca**/**Dealloc**) during its execution. This activates the **Dynamically Allocated Data Protection** composed by the connections ④ and ⑤. This request is received by the **Allocator or Deallocator**. Before perform the operation, it communicates to the **O.S. Kernel** which is the operation to trace. When the **O.S. Kernel** ends in updating its metadata with the new operation answers back to the **Allocator or Deallocator** that will perform the operation and forward the result to the **Running Program** (**Chunk**/**Nothing**).

4.6 Authorization Mechanism

In a real scenario, applications change execution status from a privilege level to another. Therefore, I introduce an authorization mechanism that verifies the profile to allow the O.S. to grant or deny a change request.

I consider it as a separate module because it could be also a third-party application interfaces to the O.S. or, in general, there could be different kinds of authentication methods working together. The implementation may range from a the simple password to more complex factors like smart cards.

An important part is how this module is invoked by the O.S.. The first and basic way to do that is to let the application ask the O.S. to invoke the authentication. The second method is to change the O.S. in such a way that it could be able to automatically detect the changes and invoke the authentication. In my implementation I use the former method because I want to keep my system as lightweight as possible, providing security without interfering with the application execution.

Another decision that must be taken in this module is what to do when the authentication fails. I can either close the application or redirect the application to a recovery function where the status of the application is received and continued. I decided to allow a developer to define a clean exit point for the application. The latter option is more robust and elegant but rather complex, so from the developer point of view it would decrease the overall usability of PRIVMUL.

As shown in Figure 4.5, the three round trip connections from the running application to the user, here called ③,⑥ and ⑦, represent the authorization mechanism. The application asks the O.S. (**Privilege Request**) to switch to a different profile. The O.S. activates an authentication daemon (**Auth. Request**), which prompts the user to enter (**Token Request**) a proper authentication token (password, smart card). Depending on the correctness of the token (**Token Input**), a **Success/Fail** answer is sent to the O.S., which will either allow or deny the user-profile switching.

4.7 Dynamically Allocated Data Protection

This module could be either integrated with the O.S. or an external component. The idea here is that this module receives information for each dynamic allocation and deallocation and it keeps track of every single chunk of memory of the application.

Each chunk is assigned to a privilege level, according to proper privilege level policies. One example of policy could be to assign to each newly allocated chunk the privilege level of the profile that is active during the allocation. Other more sophisticated methods could rely on static analysis to deduct the best level.

Elements ④ and ⑤ in Figure 4.5 explain how an allocation and deallocation is received (**Alloc**/**Dealloc**) and stored by the tracking mechanism that then is designated to forward the request to be completed to the system (**Alloc**/**Dealloc Trace**).

Chapter 5

Proposed Implementation

In this section I describe the proof-of-concept implementation of PRIVMUL in details.

I developed PRIVMUL using the onion scheme (see Figure 4.2). I implemented the binary preparation inside the LLVM/Clang compiler infrastructure. I added three transformation passes to the compiler chain. I modified Glibc (dynamic loader and dynamic memory allocator) and Linux kernel, adding three system calls and one module. Furthermore, I implemented a daemon for the authentication that uses PAM libraries to support different authentication methods. Along with an high level descriptions, during this chapter I provide a real example to show the PRIVMUL usage on a small program which code is reported in Listing 5.1.

Listing 5.1: Running Example Source Code

```
#include <iostream>
int fun1() __attribute__((privilegeSeparation(3)));
int fun2() __attribute__((privilegeSeparation(5)));
int fun3();      /* NON TAGGED FUNCTION */
int fun4() __attribute__((privilegeSeparation(4)));
int __attribute__((privilegeSeparation(5))) shared = 0;
int fun1()
{
    shared = 1;
    fun3();
    return shared;
}
int fun2()
{
    shared = 2;
    fun3();
    fun4();
    return shared;
}
int fun3()
```

```
{
    shared = 3;
    return shared;
}
int fun4()
{
    return 5;
}
int main(void)
{
    int res1 = fun2();
    int res2 = fun1();
    std::cout << res1 << ", " << res2 << std::endl;
    return 0;
}
```

5.1 Compile Time

All the transformations depend on the annotations provided by the developers. I implemented the transformation passes inside the LLVM/Clang compiler infrastructure.

5.1.1 Tagging Mechanism

The tagging mechanism defines a new attribute that allows developers to express the least accessible level for a function or a global variable. At compile time, the new attribute is processed by Clang (compiler front-end), which already supports attributes by forwarding their name to the subsequent pass or LLVM along side the associated function or G.V. identifier.

Since I need to keep track of the attribute parameter, I modified Clang to forward the parameter value to LLVM. The attribute is considered valid for both functions and global variables declarations and specifies the privilege level of functions and global variables. For instance, Listing 5.2 exemplifies a function `f()` of privilege level 2 and a global variable at privilege level 6. In this example, the developer wants to prevent the code of function `f()` from accessing the memory area where the variable `e` will be stored at runtime.

Listing 5.2: Attribute Function and Global Variable Usage Example

```
void f() __attribute__((privilegeSeparation(2)));
int e __attribute__((privilegeSeparation(6)));
```

The parameter indicates the privilege level which ranges between 0 and a tunable constant

called NUM_OF_LEVELS. This particular level is considered accessible by all the other levels. The `main()` function is automatically set to NUM_OF_LEVELS by PRIVMUL to let the program start. If the developer tries to use a greater level of NUM_OF_LEVELS, he or she receives a compile time error.

5.1.2 Tagging Propagation

This is the first transformation pass. It analyzes attribute values specified by the developers and propagates them to the non-tagged functions. Clearly, the developer can tag every function and G.V.. However, it is better to limit the effort of the developer to the bare minimum and let PRIVMUL's modified compiler do the rest in order to minimize the chances of introducing errors.

First this pass finds the Call Graph (C.G.) nodes that represent the `main()` function (the root) and then walks the C.G. breath first. Every tagged node are expanded, putting the children in the list of non visited nodes, continuing until the list is empty. For each non tagged node it checks if all parent nodes have already been tagged. In this case the algorithm tries to find the most accessible level among all the parents and use it to tag the child (see example in Figure 5.1). On the other hand, if not all the parents have been tagged, the node is put at the end of the list of non visited nodes. When the list of non visited nodes is empty the algorithm checks that all the nodes have the correct tag. If it finds one node without a tag it restarts the analysis from the root node. It runs till all the nodes have a tag. When the algorithm detects that the propagation is not converging, it aborts the compilation. In this case the developers must provide some more tags to help the algorithm in its operations.

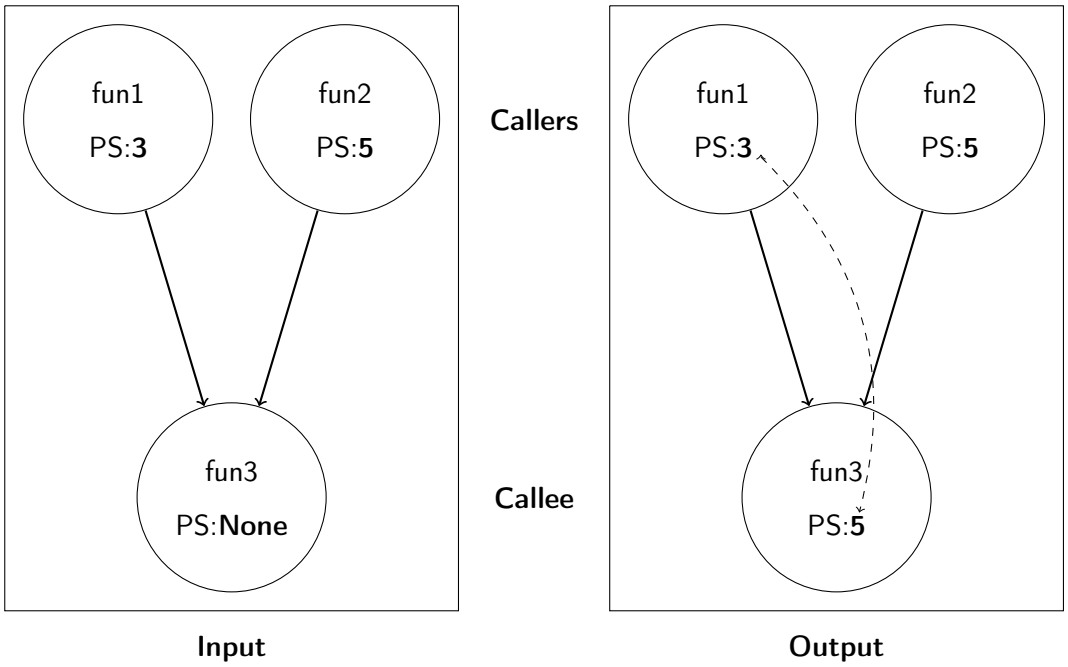

Figure 5.1: Tags propagation on a toy call graph example. Between the levels 3 and 5 the algorithm choose 5 for the non-tagged function `fun3` because it is the most accessible between the two. Functions of level 3 are still able to see the code of `fun3` as showed by the dashed arrow.

Listing 5.3: Compilation output of the real example. In this output it is notable the tag assigned to the `fun3()` (non tagged in the source code) that is called by `fun1()` and `fun2()` with respectively privilege levels 3 and 5. The algorithm chose to propagate 5 because it is the most accessible between the two.

```
------------------TAGS PROPAGATION-----------------------
Function _{cxx_global_var_init tag .text.startup
Function _Z4fun1v tag .fun_ps_3
Function _Z4fun3v tag .fun_ps_5               <-- fun3()
Function _Z4fun2v tag .fun_ps_5
Function _Z4fun4v tag .fun_ps_4
Function main tag .fun_ps_9
Function _GLOBAL_I_a tag .text.startup
---------------SYSTEM CALL INSERTION---------
Returning callsite: Caller =  _Z4fun1v Callee = _Z4fun3v
Returning callsite: Caller =  _Z4fun2v Callee = _Z4fun3v
Returning callsite: Caller =  _Z4fun2v Callee = _Z4fun4v
Returning callsite: Caller =  main Callee = _Z4fun2v
Returning callsite: Caller =  main Callee = _Z4fun1v
```

5.1.3 System Calls Insertion and Error Handler

PRIVMUL protects code and data of all the levels that are higher than the current level, but, at the same time, allows a certain flexibility during the execution of the program. The application should be able to switch level at runtime. To do so, PRIVMUL leverages a new system call to inform the Linux kernel that the application requested for a level switch. This system call presents in every call site of the program when the caller has a lower level of execution than the callee. After the call site there should be a second system call to inform the kernel that the application should release the privileges acquired with the previous. To avoid the manual intervention of the developers, I provide an automatic algorithm, so as to avoid any mistakes, making the adoption of PRIVMUL as easy as possible. To this end, whenever this transformation pass encounters a call site, it inserts an if-like snippet like the one exemplified in Listing 5.4. A real example is exemplified in Listing 5.5 where a function of level 5 is asking to move to level 4.

Listing 5.4: Error Handler Usage Example

```
if (syscall_upgrade() != 1) {
    exit_wrapper();
}
call();
syscall_downgrade();
```

Listing 5.5: Disassembled binary with system call insertion: these assembly lines are taken from the compiled binary of the real example presented. They represent a piece of the `fun2()` where `fun4()` is called since `fun4()` belongs to level 4 and `fun2()` to level 5, to be able to execute `fun4()` I need more privileges. For this reason the if-like structure described above is automatically inserted by the compiler pass. The result is as follow:

```
movl   $0x4,0x4(%esp)            <--
movl   $0x160,(%esp)               |  syscall_upgrade()
call   8048640 <syscall@plt>     <--  to level 4

cmp    $0x1,%eax                 <--  if syscall res is equal
je     804b8ba <_Z4fun2v+0x3a>   <--  to 1 jump to call()
                                      otherwise go to exit()

movl   $0xffffffff,(%esp)
call   80486c0 <exit@plt>        <--  exit(-1)

call   804a870 <_Z4fun4v>        <--  call();

movl   $0x5,0x4(%esp)            <--
movl   $0x160,(%esp)               |  syscall_downgrade()
call   8048640 <syscall@plt>     <--  to level 5
```

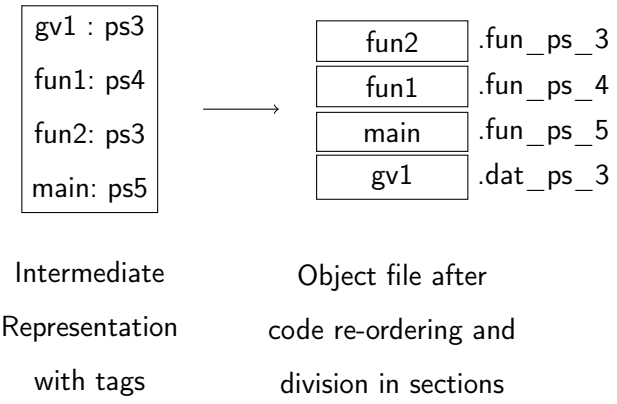

Figure 5.2: Transformation passes effect on the intermediate representation

In case of failure because the user is not eligible for the requested level, a wrapper is called. PRIVMUL allows the developer to specify his/her own wrapper with the name `exit_wrapper`, where he or she can decide to either continue the application recovering its status or let the program exit. If nothing is specified from the developer, the system will substitute that specific call with a direct call to `exit(-1)` avoiding segmentation fault.

5.1.4 Code and Data Re-ordering

The last code modification is the re-ordering of all the functions. Every piece of code and data is, at this point, tagged with a certain level. With a very simple LLVM pass each function and each global variable are ordered in an ascending fashion way according to the privilege level. Then, each element is inserted in a new section of the binary. To this end I exploit the ELF format, which allows the creation and insertion of new non standard sections. The name of the section will be created merging the level number and either a function or a global variable. In the case of function the name will be like `fun_ps_X` for every function that will belong to the level X. The same occurs for data, with creating something like `dat_ps_X`. This pass also keeps track of all the levels used by the program under analysis and generates and writes a custom linker script into the same folder. In the next section, the custom linker script will be explained in details.

5.2 Link Time

In this phase PRIVMUL has created an object file and a linker script, which is used to generate a new segment for each **fun_ps** or **dat_ps** section in the final binary. This partitioning at segment level is necessary to forward in the process all the information about which area of memory belongs to each level of privilege. Furthermore, PRIVMUL, after the binding between segments and sections, aligns the start address of each segment to the beginning of the next page of memory (see example in Figure 5.2). This is necessary because otherwise the kernel sees a wrong layout when the application is launched, because one or more segments are merged together when the binary is mapped in memory. This generates wrong operations during the runtime.

Listing 5.6: Physical Header Overwrite

```
PHDRS
   {
    headers PT_PHDR PHDRS ;
    interp PT_INTERP ;
    text PT_LOAD FILEHDR PHDRS ;
    fun_ps_3 PT_LOAD ;
    fun_ps_4 PT_LOAD ;
    fun_ps_5 PT_LOAD ;
    fun_ps_9 PT_LOAD ;
    data PT_LOAD ;
    dat_ps_5 PT_LOAD ;
    dat_ps_9 PT_LOAD ;
    dynamic PT_DYNAMIC ;
   }
```

Listing 5.7: Physical header overwrite effect on a real binary: looking into the binary header it is possible to inspect the effects of PRIVMUL on the real application. In program headers are visible more LOAD segments than a normal binary where usually the LOAD segments are two (code and data). The second part shows the section to segment mapping, it shows which are the sections contained in the LOAD segments reported in the first part of the following listing.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x08048194 0x08048194 0x00034 0x00033 R   0x4
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x05a5c 0x05a5c R E 0x1000
  LOAD           0x006850 0x08049850 0x08049850 0x0001f 0x0001f R E 0x1000
  LOAD           0x006870 0x0804a870 0x0804a870 0x0000a 0x0000a R E 0x1000
  LOAD           0x006880 0x0804b880 0x0804b880 0x0005d 0x0005d R E 0x1000
  LOAD           0x0068e0 0x0804c8e0 0x0804c8e0 0x00140 0x00140 R E 0x1000
  LOAD           0x006ee4 0x0804eee4 0x0804eee4 0x0015c 0x0320c RW  0x1000
```

```
  LOAD           0x007040 0x08050040 0x08050040 0x00004 0x00004 RW  0x1000
  LOAD           0x007044 0x08051044 0x08051044 0x00004 0x00004 RW  0x1000
  DYNAMIC        0x006ef4 0x0804eef4 0x0804eef4 0x00144 0x00144 RW  0x4

Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version
       .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr
       .eh_frame
   03     .fun_ps_3
   04     .fun_ps_4
   05     .fun_ps_5
   06     .fun_ps_9
   07     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   08     .dat_ps_5
   09     .dat_ps_9
   10     .dynamic .got .got.plt
```

Listing 5.8: Linker script example: This script parts are automatically inserted in the standard script of ld (the GNU linker) when the application is compiled. The number of elements added to the script depends on the number of different privilege levels used by the application. The line `. = . 0x1000;` added at the end of each privilege level block is used to move the elements that follow in other pages of memory.

```
1  [...]
2  .text :
3    {
4      *(.text.unlikely .text.*_unlikely)
5      *(.text.exit .text.exit.*)
6      *(.text.startup .text.startup.*)
7      *(.text.hot .text.hot.*)
8      *(.text .stub .text.* .gnu.linkonce.t.*)
9      /* .gnu.warning sections are handled specially by elf32.em.  */
10     *(.gnu.warning)
11   } : text
12  . = . + 0x1000;
13 /* ADDITIONAL ELEMENTS FOR FUNCTION SECTIONS ALIGNMENT */
14 .fun_ps_3 :
15   {
16    *(fun_ps_3)
17   } : fun_ps_3
18 . = . + 0x1000;
19 [...the same for level 4 and 5]
20 .fun_ps_9 :
21   {
22    *(fun_ps_9)
23   } : fun_ps_9
24   . = . + CONSTANT (COMMONPAGESIZE) - SIZEOF (.fun_ps_9);
25 /*********************************************/
26 [...]
27 .data :
28   {
```

```
29      *(.data .data.* .gnu.linkonce.d.*)
30      SORT(CONSTRUCTORS)
31    } : data
32  . = . + 0x1000;
33
34
35
36  /* ADDITIONAL ELEMENTS FOR DATA SECTIONS ALIGNMENT */
37  .dat_ps_5 :
38    {
39      *(dat_ps_5)
40    } : dat_ps_5
41  . = . + 0x1000;
42  .dat_ps_9 :
43    {
44      *(dat_ps_9)
45    } : dat_ps_9
46    . = . + CONSTANT (COMMONPAGESIZE) - SIZEOF(.dat_ps_9);
47  /*********************************************/
48  [...]
```

The code in Listing 5.6 is the responsible for the section-segment binding. More precisely it defines the `program headers`, which describe how the program should be loaded into memory (see real ELF Header case in Listing 5.7). Usually, the code in Listing 5.6 is not specified in the normal script used by the linker but to achieve a finer customization of the binary I had to use it.

In Listing 5.8, I can identify the two additional pieces I inserted to align all the sections. On one hand, from the line 13 to the 25 is showed the function sections alignment. This piece is placed right below the `.text` block that usually it is the only one specified for the code of the application. On the other hand, from the line 36 to 47 I can see the global variable sections alignment. This block is placed right below the .data block where usually almost all the global variable are placed in a standard binary. As I have already mentioned, every additional elements inside the link script is automatically generated from the information found inside the source code of the application.

5.3 Run Time

At this point, the binary has been produced and is ready to be run. The next subsections explain the runtime execution in details. Section 5.3.1 explains how the O.S. is instrumented by the application before it is executed. The other Section 5.3.2 showes how the application,

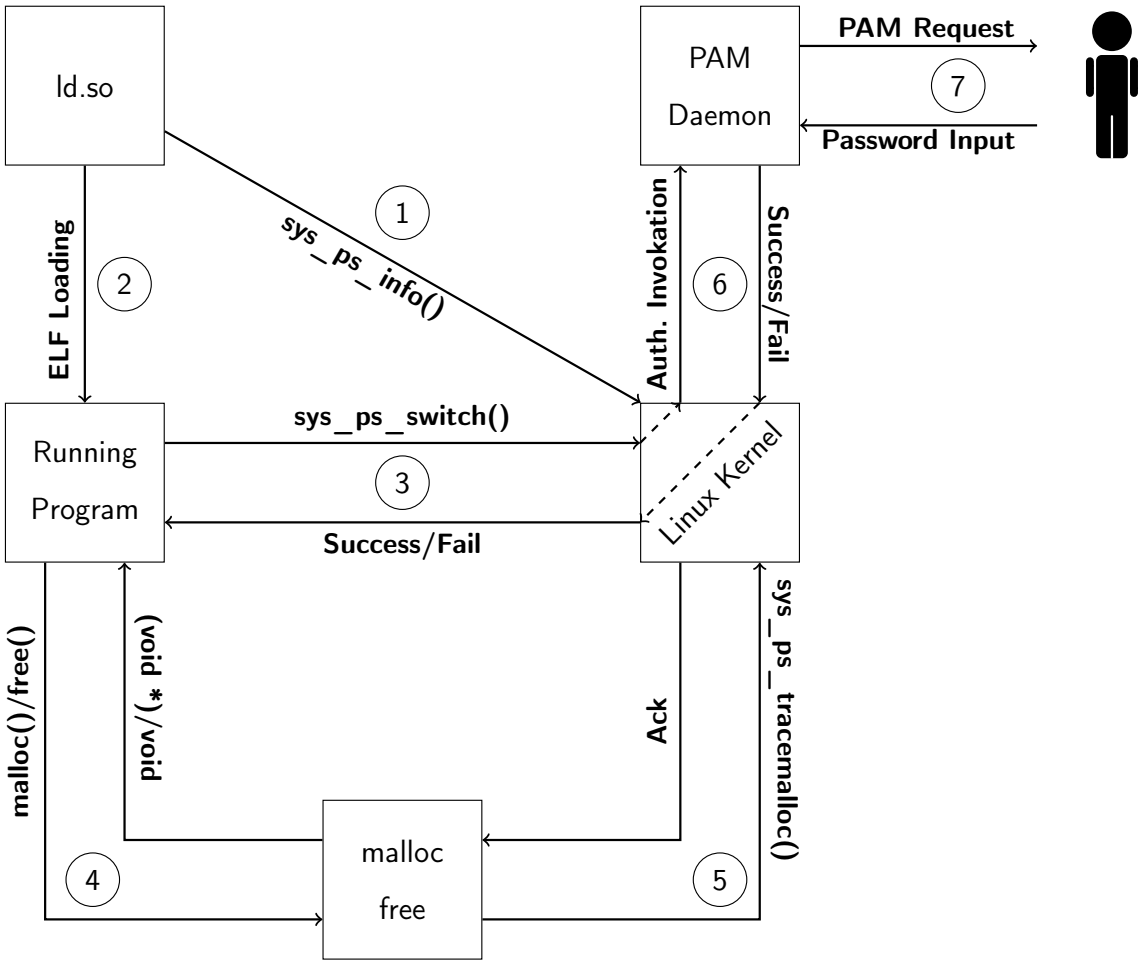

Figure 5.3: PRIVMUL implementation in the Linux environment

communicating with the O.S., is able to be executed safety (Figure 5.3).

5.3.1 Dynamic Loader

Before the first instruction of a PRIVMUL-modified binary is executed lots of things take place. This phase is called load time. I hereby describe briefly what usually happens in this phase and then I show how I made the kernel aware of the memory application layout required by PRIVMUL.

When an application is run, the kernel, that receives the request, passes control to `ld.so` (or `ld-linux.so.2`), which takes care of shared libraries (if any are used). This implies that all the shared libraries addresses must be resolved in the in-memory image of the application

before the entry-point receives the control. Because the libraries addresses may vary from one execution to execution (e.g., version changes, security mechanisms such as control-flow integrity checks or address space layout randomization).

Moreover, after the dynamic linking phase the `ld.so` starts to map all the LOAD segments of the application in memory according to the information in the ELF header. Only when all the segments are mapped, the application is ready to be executed; at this point ld.so gives the control to the real entry point of the application.

Therefore, I used `ld.so` to propagate the memory layout information from the binary to the process, because it is always executed before the application and it shares the same PID. Furthermore, it is already responsible for parsing the ELF header to load the segments from the segments table header. To this end, I extended the header parsing as follows. Segments are unnamed so I needed the name of the section contained in the segment because the name indicates which is the privilege level contained. This metadata is important for the kernel to perform its operations. I used the existing pointer to the header file to obtain all the information related to the section header. Then, comparing section and segment start and end addresses, I reconstruct section-segment binding. Finally, I constructed a linked list of `PrivSec_t` elements, which is the list of elements that needs to be sent to the kernel.

Listing 5.9: Section-segment mapping element

```
struct PrivSec_t{
    char name[100];
    Elf32_Addr add_beg;
    Elf32_Addr add_end;
    struct PrivSec_t *next;
};
```

The struct in Listing 5.9 is composed by:

- **name**: the string that contains the name of the section. It would be something like fun_ps_X or dat_ps_X.
- **add_beg**, **add_end**: the begin and end addresses of the segment. They will be used by the kernel to mprotect a segment when I need to protect it from unauthorized access.
- **next**: pointer to the next element in the linked list.

Referring to Figure 5.3, the **sys_ps_switch** is the first routine that executes when PRIV-

MUL starts the application. It sends the layout read by the loader to the kernel. Instead, the **ELF loading** arrow indicates when the loader, after all the dynamic libraries addresses have been resolved, passes the control to the application.

5.3.2 Linux Kernel

In my implementation I decided to use the Linux kernel because it is very well documented and offers plenty of interfaces. I implemented several new APIs (system calls) to allow the interaction between the application and the O.S.. From hereinafter I explain the goals and implementation of each element.

System calls

The interaction of the application with the kernel is done through three system calls.

The first one is `ps_info`.

Listing 5.10: System Call ps_info Prototype

```
asmlinkage long sys_ps_info (struct PrivSec_t *h, int level)
```

This function copies the memory layout from user space to kernel space, and attaches a list of `PrivSec_t` to the `task_struct` of the application. The `task_struct` of an application is the image representation of the running application inside the kernel. It contains all the information regarding the process (e.g., `pid`, `gid`, `locks`).

Listing 5.11: Memory Chunk Information Element

```
struct PrivSec_dyn_t{
    int ps_level;
    int size;
    void *mem;
    struct PrivSec_dyn_t *next;
};
```

The `sys_ps_info()` is executed in `ld.so` before the control is passed to the application. I decided to put this operation inside `ld.so` because it was easy to collect all the information about the layout and, at this point, the information provided could not have been tampered by an attacker. For obvious security reasons PRIVMUL allows this system call to be invoked only once for each process. Indeed, if an attacker could use this system call during the execution,

he or she would be able to obtain access to all the code and data segments by simply pushing new information to the kernel. For the same reason, after the execution of `ps_info`, also `mprotect` is locked and not executable anymore from the process.

The second system call added to the Linux kernel is called `ps_switch`.

Listing 5.12: System Call ps_switch Prototype

```
asmlinkage long sys_ps_switch(int new_level)
```

It allows an application to request the a switch from the current level to another, specified the unsigned integer parameter `new_level`.

When switching to a higher level, the kernel wakes up a daemon that blocks the application code reativation until the user has entered the credentials. In the mean time the process is moved by the O.S. into the wait state and it will be waked up only at the end of the `ps_switch` system call. If the authentication succeeds, the kernel unlocks all the respective code and memory areas and returns the control to the application. Otherwise, the program exits using a wrapper (either written by the developer or automatically inserted by PRIVMUL). I report in Listing 5.13 the status of the memory mapping of the running example taken during the execution of each function.

Listing 5.13: Memory mapping status: the reported lines are directly taken from the procfs interface of the kernel (`/proc/PID_REAL_EXAMPLE/maps`) in different moment of the application execution.

```
#In the main function at the beginning of the program most of the segments
#are not accessible (level 9).
08048000-08049000 r-xp 00000000 08:03 3408569    /root/test-application/prova
08049000-0804a000 ---p 00006000 08:03 3408569    /root/test-application/prova  <<
0804a000-0804b000 ---p 00006000 08:03 3408569    /root/test-application/prova  <<
0804b000-0804c000 ---p 00006000 08:03 3408569    /root/test-application/prova  <<
0804c000-0804d000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804d000-0804e000 r-xp 00005000 08:03 3408569    /root/test-application/prova
0804e000-08050000 rw-p 00006000 08:03 3408569    /root/test-application/prova
08050000-08051000 ---p 00007000 08:03 3408569    /root/test-application/prova  <<
08051000-08052000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08052000-08053000 rw-p 00000000 00:00 0
b74a4000-b74a7000 rw-p 00000000 00:00 0
b74a7000-b7653000 r-xp 00000000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7653000-b7655000 r--p 001ac000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7655000-b7656000 rw-p 001ae000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7656000-b7659000 rw-p 00000000 00:00 0
[...]

#In the fun2 the program is running as level 5, the segments for the fun2 and
#the shared variable are now accessible.
08048000-08049000 r-xp 00000000 08:03 3408569    /root/test-application/prova
```

```
08049000-0804a000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804a000-0804b000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804b000-0804c000 r-xp 00006000 08:03 3408569    /root/test-application/prova  <<
0804c000-0804d000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804d000-0804e000 r-xp 00005000 08:03 3408569    /root/test-application/prova
0804e000-08050000 rw-p 00006000 08:03 3408569    /root/test-application/prova
08050000-08051000 rw-p 00007000 08:03 3408569    /root/test-application/prova  <<
08051000-08052000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08052000-08053000 rw-p 00000000 00:00 0
b74a4000-b74a7000 rw-p 00000000 00:00 0
b74a7000-b7653000 r-xp 00000000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7653000-b7655000 r--p 001ac000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7655000-b7656000 rw-p 001ae000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7656000-b7659000 rw-p 00000000 00:00 0
[...]

#From fun2 is called fun3 with the same privilege level and fun4 that belongs
#to level 4. Its segment is switched to accessible along with level 5 segments
#because level 5 is accessible by level 4 in the onion scheme.
08048000-08049000 r-xp 00000000 08:03 3408569    /root/test-application/prova
08049000-0804a000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804a000-0804b000 r-xp 00006000 08:03 3408569    /root/test-application/prova  <<
0804b000-0804c000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804c000-0804d000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804d000-0804e000 r-xp 00005000 08:03 3408569    /root/test-application/prova
0804e000-08050000 rw-p 00006000 08:03 3408569    /root/test-application/prova
08050000-08051000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08051000-08052000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08052000-08053000 rw-p 00000000 00:00 0
b74a4000-b74a7000 rw-p 00000000 00:00 0
b74a7000-b7653000 r-xp 00000000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7653000-b7655000 r--p 001ac000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7655000-b7656000 rw-p 001ae000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7656000-b7659000 rw-p 00000000 00:00 0
[...]

#After fun4 the execution comes back to the main and the mapping accordingly changes
#to level 9 again stepping through from level 4 to 5 and then from level 5 to 9.
08048000-08049000 r-xp 00000000 08:03 3408569    /root/test-application/prova
08049000-0804a000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804a000-0804b000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804b000-0804c000 ---p 00006000 08:03 3408569    /root/test-application/prova
0804c000-0804d000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804d000-0804e000 r-xp 00005000 08:03 3408569    /root/test-application/prova
0804e000-08050000 rw-p 00006000 08:03 3408569    /root/test-application/prova
08050000-08051000 ---p 00007000 08:03 3408569    /root/test-application/prova
08051000-08052000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08052000-08053000 rw-p 00000000 00:00 0
b74a4000-b74a7000 rw-p 00000000 00:00 0
b74a7000-b7653000 r-xp 00000000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7653000-b7655000 r--p 001ac000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7655000-b7656000 rw-p 001ae000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7656000-b7659000 rw-p 00000000 00:00 0
[...]

#At this point of the execution the privileged call to fun1 is performed. The mapping
```

```
#is changed again and it reaches level3 that in our example is the outer layer of the
#onion that means it can access all the other levels.
08048000-08049000 r-xp 00000000 08:03 3408569    /root/test-application/prova
08049000-0804a000 r-xp 00006000 08:03 3408569    /root/test-application/prova  <<
0804a000-0804b000 r-xp 00006000 08:03 3408569    /root/test-application/prova  <<
0804b000-0804c000 r-xp 00006000 08:03 3408569    /root/test-application/prova  <<
0804c000-0804d000 r-xp 00006000 08:03 3408569    /root/test-application/prova
0804d000-0804e000 r-xp 00005000 08:03 3408569    /root/test-application/prova
0804e000-08050000 rw-p 00006000 08:03 3408569    /root/test-application/prova
08050000-08051000 rw-p 00007000 08:03 3408569    /root/test-application/prova  <<
08051000-08052000 rw-p 00007000 08:03 3408569    /root/test-application/prova
08052000-08053000 rw-p 00000000 00:00 0
b74a4000-b74a7000 rw-p 00000000 00:00 0
b74a7000-b7653000 r-xp 00000000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7653000-b7655000 r--p 001ac000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7655000-b7656000 rw-p 001ae000 08:03 5384798    /usr/src/root-glibc/lib/libc-2.18.so
b7656000-b7659000 rw-p 00000000 00:00 0
[...]
```

The third system call is the `ps_tracemalloc` Listing 5.14, which manages dynamic memory allocations. Typically, dynamic memory is allocated at runtime by functions such as `malloc` or `valloc`, destroyed by functions such as `free`, and modified by functions such as `realloc`.

Listing 5.14: System Call ps_tracemalloc Prototype

```
asmlinkage long sys_ps_tracemalloc(void *ptr, int size, char *cmd)
```

I prepended these functions in the Glibc libraries with this system call such that every time new memory is requested the kernel saves the newly allocated memory area in a specific list attached to `task_struct` of the process. Similarly, I instrumented free-like functions such that the kernel removes the memory from the list of chunks under protection. In the case where a certain area is changed, the operations kernel side are not always necessary for the allocator itself, instead PRIVMUL has always to perform this system call to keep the metadata updated. The PrivSec_dyn_t data structure, detailed in Listing 5.11, is used in the kernel to keep track of dynamically allocated memory areas. The `ps_level` field represents the current privilege level when the area of memory is requested. If a piece of data created by level n cannot be accessed by a level m unless $m \geq n$. The second field of the structure is the size of the chunk. This is useful to be protected or unprotected at runtime. The third field in the struct is the raw pointer to the memory area. The fourth element is the pointer to the next element in the list.

Authentication and Authorization

Referring to Figure 5.3, ③ together with the ⑥ and ⑦ groups represent the authorization mechanism. The implementation of the authorization is composed by a user space daemon (its behavior depends on the authentication method performed), a kernel netlink interface and the system call described in the previous section. Before the application is executed, the netlink interface is activated through a LKM and the daemon starts. This could be done manually or automatically when needed. When the application calls the system call to switch privilege level, the system call writes into the netlink channel (where the daemon is waiting for data) the privilege level that the application wants to switch to. The daemon starts a request of authentication for that level using the PAM subsystem. Then, it writes back to the system call through the netlink interface an appropriate message indicating whether or not the authentication has been granted. At this point, the system call thread is waked up to return the result to the program. According to the result, the application will gracefully exit or continue. The netlink communication channel is secure according to the attacker model defined in Section 4.1 (i.e., attackers without root access to the machine).

The arrows labeled as ⑥ in Figure 5.3 the netlink interface that communicates with the PAM daemon from the kernel. Instead, the arrow labeled as ⑦ indicates the communication between the PAM daemon and the user through the standard input and output.

I decided to use PAM because it is very well tested and widely used. PAM is modular, very versatile, and supports several combinations of authentication mechanisms (e.g., passwords, smart cards, fingerprints etc.). For instance Listing 5.15 shows how a password based authentication can be enabled by editing `/etc/pam.d/ps_login`.

Listing 5.15: PAM module configuration example

```
# /etc/pam.d/ps_login

auth      required    pam_unix.so shadow nullok
account   required    pam_unix.so
session   required    pam_unix.so
password  required    pam_cracklib.so retry=3
```

Chapter 6

Experimental Results

In the evaluation of PRIVMUL I quantified the overhead of PRIVMUL, to evaluate its impact on the overall usability. Regarding the authorization mechanism, I can conclude that the real impact depends on the number of operations performed by a function. If the function has few instructions, the percentage of the overhead is considerably high; on the other hand, if the number of instructions computed is around 900,000 or higher, the overhead is acceptable. Regarding the dinamically allocated memory tracking mechanism I can conclude that the slowdown on average introduced is about 8x (4x in the best case and 11x in the worst case, according to my measurements).

6.1 Goals

I created two tests to evaluate the performance losses, one for each component of PRIVMUL that could induce runtime overhead to the instrumented application. I can safely ignore compile-time overhead and the load-time overhead as they are paid only once.

- **Authorization phase:** I measured the execution time of a function with and without PRIVMUL.
- **Dynamic memory allocation:** I measured the execution time of a function call that made intense use of the dynamically allocated data, with and without PRIVMUL

To measure the exact time of the system call execution, avoiding the overhead due to the `clock()` library function, which uses a system call to obtain the clock information I computed

time directly from the tick counter register, which is included in all modern Intel processors. To this end, I used the assembly instructions reported in Listing 6.1 and 6.2.

Listing 6.1: CPU Tick Count Start

```
static inline uint64_t cycle_start(void)
{
        uint32_t cycles_low, cycles_high;

        asm volatile (
                "cpuid\n"
                "rdtsc\n"
                "movl %%eax, %0\n"
                "movl %%edx, %1\n"
                : "=r" (cycles_low), "=r" (cycles_high)
                :
                : "%rax", "%rbx", "%rcx", "%rdx"
        );

        return (uint64_t) cycles_high << 32 | (uint64_t) cycles_low;
}
```

Listing 6.2: CPU Tick Count Stop

```
static inline uint64_t cycle_stop(void)
{
        uint32_t cycles_low, cycles_high;

        asm volatile (
                "rdtscp\n"
                "movl %%eax, %0\n"
                "movl %%edx, %1\n"
                "cpuid\n"
                : "=r" (cycles_low), "=r" (cycles_high)
                :
                : "%rax", "%rbx", "%rcx", "%rdx"
        );

        return (uint64_t) cycles_high << 32 | (uint64_t) cycles_low;
}
```

I instrumented the `sys_ps_switch()` system call to make the measure. The mean time for the system call is 560370.33 CPU cycles when an upgrade is requested (on 1000 times repetition). This cycles number means, with a 3.6 Ghz processor, 155,65μs. On the other hand, the mean time for the system call is 6882.66 CPU cycles when a downgrade is requested. With a 3.6 Ghz processor, it means 1,96μs. The reader should keep in mind that the gap that is in the graphs includes the context switch time between user space and kernel space that is not kept into account in the second measure of this test.

6.2 Experiment Setup

I made all the experiments in a fully virtualized machine (VirtualBox) running a Gentoo Linux guest with the PRIVMUL-patched kernel (version 3.9.11) and Glibc, with the modified loader and tracking mechanism (stable version 2.18). The machine was also instrumented with a LLVM/Clang compiler infrastructure (version 3.5) with the inclusion of my transformation passes. The virtual machine had 2 i7-3520M cores with 3.60 GHz (max turbo frequency) and 4GB of RAM.

I recorded two user profiles as system users and I set a password for each of them.

Clearly, my measurements did not include the interaction time (i.e., time required by the user to type the password), as this varies from user to user, and is influenced by the authentication method employed. To this end, I slightly modified the authentication daemon such that the authentication was always successful without any password input. All the tests described were single-process tests. In all the executions, to avoid the introduction of random switch of core due to scheduler decision, I bound each process (using the set affinity) to one core.

6.3 Experiment 1: Authentication and Authorization

In this test, for a single switch I wanted to show how much overhead is introduced by the authorization loop (i.e., arrows ③, ⑥, and ⑦ in Figure 5.3). I made two versions of this specific test. In the first one the number of instructions executed by the function is around 900,000. In the second one, it is around 9,000. I took the time from the point before the call to the return to the caller as the Listing 6.3 shows. I repeated the experiment 10,000 times and calculated mean and standard deviation.

Listing 6.3: Piece of code that measures the execution time for experiment 1

```
[...]
for(i=0; i<10000; ++i) {
     start = cycle_start();
     test1_priv(i);
     stop = cycle_stop();
     time_spent = (stop - start) / MAX_FREQ;
}
[...]
```

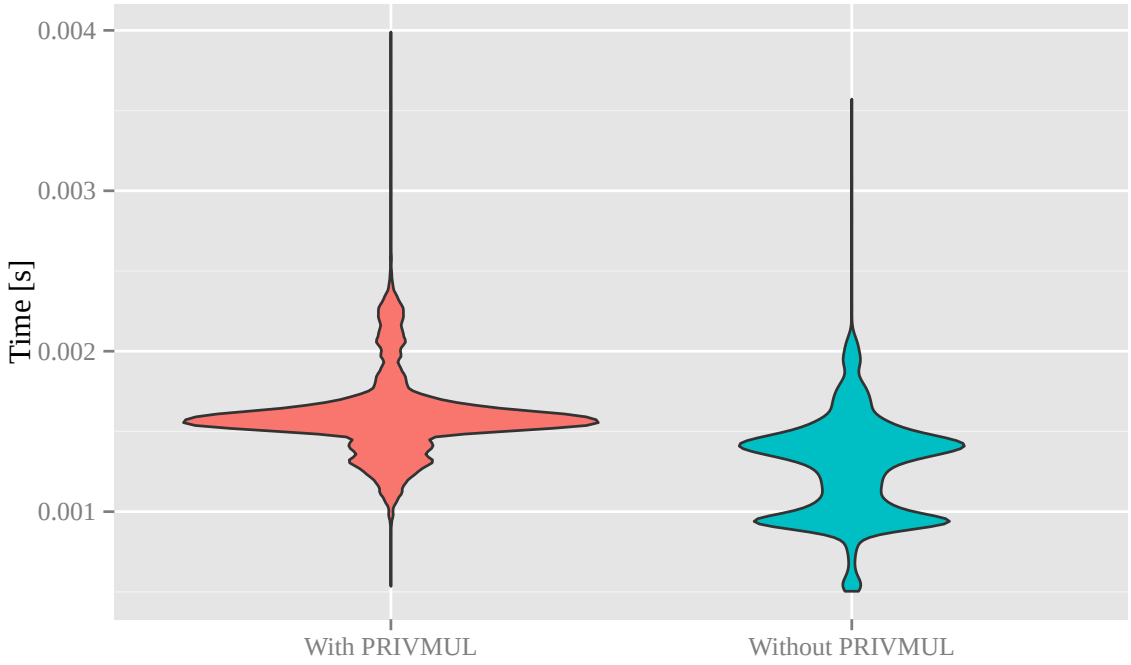

Figure 6.1: Violin plot of the test results on the long function (900,000 instructions).

By comparing Figure 6.1 and Figure 6.2 I conclude that the real impact of my system depends on the number of operations performed by the function. If the function has few instructions (Figure 6.2), the percentage of the overhead is considerably high; on the other hand, if the number of instructions computed is around 900,000 or higher (Figure 6.1), the overhead is acceptable.

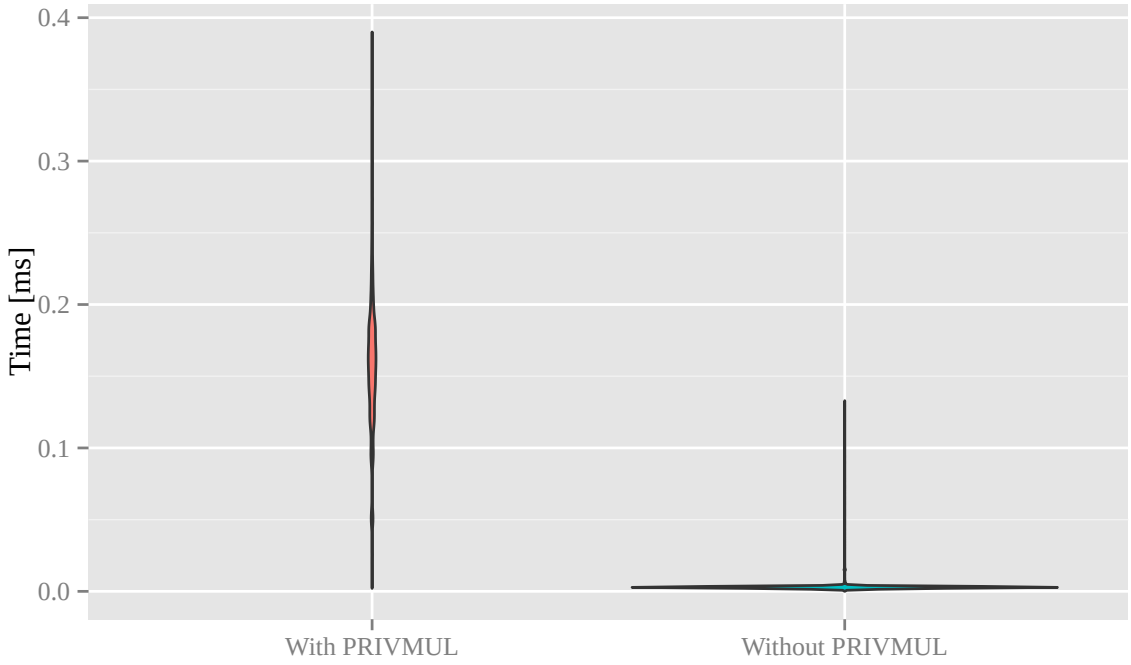

Figure 6.2: Violin plot of the test results on the short function (9,000 instructions).

6.4 Experiment 2: Dynamic Memory Allocation

In this experiment I wanted to measure the overhead of the dynamically allocated memory tracking mechanism. To this end, I wrote two tests. The first test, reported in Listing 6.4, is a function that allocates and deallocates a fixed amount of memory. In both cases I repeated the experiment 5,000 times comparing the average of the results when the application was using the instrumented libraries and the case of un-instrumented one.

Listing 6.4: First test with one allocation and deallocation

```
[...]
for(i=0; i<5000; ++i) {
    start = cycle_start();
    int *p = (int *) malloc(sizeof(int) * 1000);
    free(p);
    stop = cycle_stop();
    time_spent = (stop - start) / MAX_FREQ;
}
[...]
```

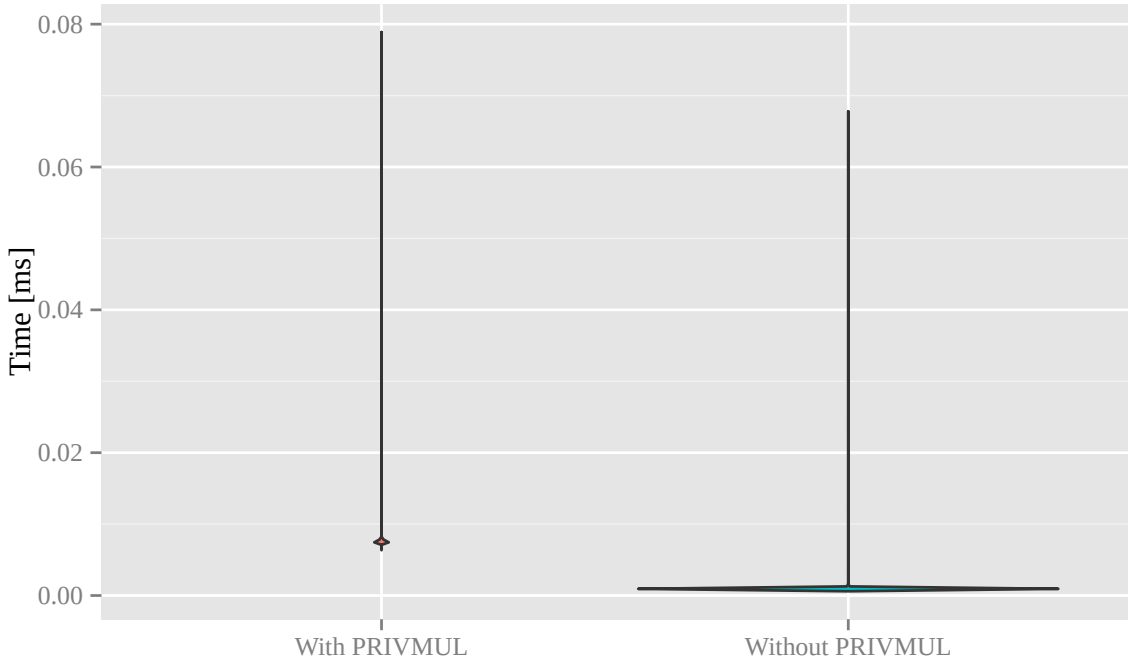

Figure 6.3: Distribution of the results of the first memory test, one allocation and deallocation of a piece of fixed amount of bytes.

The second test, reported in Listing 6.5, consists of a function where I made 500 allocations and deallocations of a different amount of memory every allocation and deallocation but the same sequence from time to time, which is for triggering the caching mechanism inside the allocator.

Listing 6.5: Second test with 500 variable-size allocations and deallocations

```
[...]
for(i=0; i<5000; ++i) {
    int j;
    start = cycle_start();
    for (j=0; j<500; ++j){
        int *p = (int *) malloc(sizeof(int) * j);
        free(p);
    }
    stop = cycle_stop();
    time_spent = (stop - start) / MAX_FREQ;
}
[...]
```

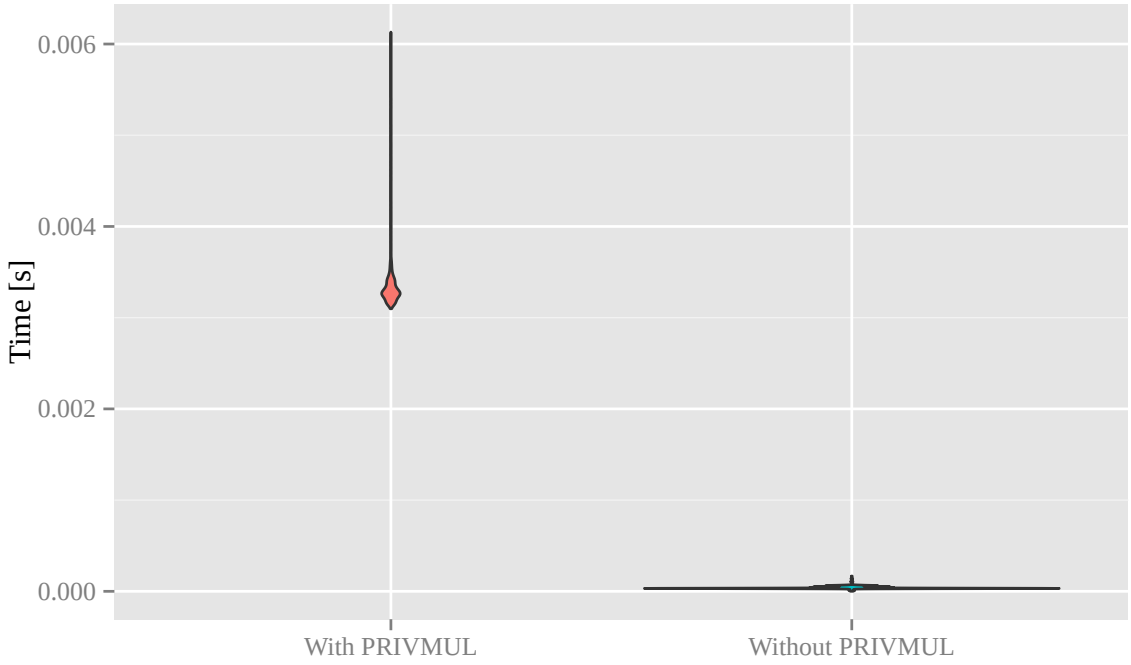

Figure 6.4: Distribution of the results of the second memory test, on 500 allocations and deallocations of different size (the sequence of allocation is the same for every run).

Figure 6.3 shows the overhead introduced by PRIVMUL (in ms), which is mainly due to the system calls that update the kernel side information. Notice that, without PRIVMUL the allocator does not release all the memory when the free is invoked. Some of the memory stays unallocated but assigned to the process to gain performance for next requests of memory by the application. So, for the normal allocator not all the operation requires requests to the kernel. With PRIVMUL, instead, in order to keep all the kernel metadata updated I needed to introduce kernel communication for almost all the cases of allocation, deallocation and resizing of memory area, which resulted in about 8x slowdown on average (4x in the best case 11x in the worst case, according to my measurements).

Figure 6.4 show the overhead on 500 allocations and deallocations. Again the kernel communication introduced for each memory operation is the main cause of the overhead - as expected.

6.5 Conclusions

PRIVMUL's overhead is significant on applications that use in-memory objects that are quickly created and destroyed. However, if I consider applications that fit the goals of PRIVMUL (i.e., large applications that make several interaction with the user and execute long functions), the overhead is acceptable (4x to 11x). Considering the effor required by the developer to secure a large application, the benefit brought forth by PRIVMUL is remarkable: At the price of a simple annotation that specifies the access level of data or code, PRIVMUL takes care of all the rest. In other words, the balance between usability vs. security vs. performance overhead of PRIVMUL is significant.

Chapter 7

Discussion and Future Work

7.1 Limitations

The advantages offered by PRIVMUL are remarkable, because it allows the developers to express access-control policies by means of simple source-code-level annotation, pretty much like it happens in object oriented languages (e.g., Java), yet with more expressive power (i.e., multi-level policies) and without requiring verbose annotations.

PRIVMUL has only two conceptual limitations, and a relatively small number of technical limitations (which are mainly bound to the current implementation).

PRIVMUL implies the availability of the source code. As stated above, the first restriction of PRIVMUL is that it is designed to work on the source code of the application. This is a problem in the case I want to secure proprietary application where the source code is not available because only the binary is provided.

Related to this source code limitation I can find another issue; most of the applications with a multi-user logic are not designed with the proper privilege level scheme in mind (such as the tree or the onion schemes discussed in 4.2). So small modifications of the program may be required to port the source code to use PRIVMUL. This change of the program may require in-depth knowledge of the codebase.

To overcome this problem would be possible if we move the tagging block forward in the process. The solution would work directly on the binary code using a mix of static analysis and user inputs to re-order the binary pieces and tag each of the functions. Working on the

binary would allow us to apply my approach in every kind of compiled application even if the source code is not available.

Secondly, PRIVMUL involves a O.S. customization. PRIVMUL assumes that there exist a trusted element to allow or deny critical operations inside the applications. In my proposed design, the trusted element is O.S.. It was the trusted block. As none of the modern O.S. had the features I needed, I had to modify the Linux kernel.

In a real case scenario of PRIVMUL I suppose to deal with enterprises systems. In those environments, being unable to use Microsoft Windows-based application may be a show stopper. However, PRIVMUL is based on concepts and facilities that are available in any modern O.S., including Microsoft Windows. Therefore, I believe that PRIVMUL can be easily ported to other systems.

To solve this problem I can move the trusted component out from the O.S.. The new trusted component could be another piece of software that is able to manage the memory of the application through a driver or some external module of the O.S.. It has to be noticed that external drivers and modules could extend the O.S. without including any modification to the core of it. I can think to also use APIs given by the O.S. such as the ReadProcessMemory/WriteProcessMemory under Microsoft Windows to bring protection from this external trusted component to the applications.

It supports only compiled language such as C and C++. I rely on compile-time modifications and some additional loader operations. The possibility of porting PRIVMUL to interpreted or partially-compiled languages such as Python, Perl or Ruby highly depends on the expressiveness of the intermediate bytecode. If it carries enough metadata (e.g., variables, functions), the concepts implemented in PRIVMUL can be easily ported. The modification that I applied in the Linux kernel would clearly need to be ported in the language interpreter. More precisely, most of the information used by PRIVMUL are collected and stored inside the binary at compile time. Then those pieces of information are loaded and moved to the kernel. In interpreted languages the compilation phase happens just before the execution thanks to the interpreter, the Just-In-Time compiler (JIT compiler) or both. This kind of compilation does not consider the overall binary but just small pieces of code (function-level, file-level or codefragment-level) that are going to be executed immediately after. This implies modifications in the interpreter in order to be able to manage the code segments.

A potential issue brought by interpreted languages is that they usually have a separated memory manager, between the application and the O.S., which mixes allocations for the application itself with the allocation of the interpreter.

It does not support full multi-thread. In PRIVMUL the protection is given by hiding the code and data of other profiles. This forces multithreading at a profile level only. The threads must be started inside a profile and they should terminate before a switch of user profile. What could happen if the application is not written with this constraint in mind is that some threads would see, at certain point of execution, their code mprotected and not accessible anymore. This is a technical limitation. Indeed, given enough time and development effort, the concepts of PRIVMUL can be extended at thread level, taking into account shared memory as belonging to threads according to the annotations specified by the developer. After all, threads are functions.

Finally, PRIVMUL is not optimal for memory intensive application.

7.2 Future Work

This section describes the main future works

In Subsection 7.2.1 I provide some details on how PRIVMUL could support the multi-threading. In Subsection 7.2.2 I present a possible work to bring a fully-automatic mechanism for the detection of privilege levels in PRIVMUL and, in Subsection 7.2.3 I briefly describe a possible application of the PRIVMUL approach to an interpreted language. Finally, in Subsection 7.2.4 I explain how to reduce the overhead brought by PRIVMUL on the application performance.

7.2.1 Multithreading Support

PRIVMUL works on single thread applications with multi-user logic. Removing this limitation would allow to apply PRIVMUL to general-purpose applications such as web-servers, where the parallel execution is common.

When a program is using multithreading the pages of memory where the code is written are in common among all the threads. To overcome this issue, to design and implement new parallel APIs would be useful. At system level, those parallel APIs should be able to allow us

to clone the pages of memory involved in the thread execution as many times as the number of threads are. In this way I would be able to have a finer grain of control on the regions of code to be protected. Techniques along this idea are implemented in lightweight hypervisors such as Xen (especially in para-virtualization mode), where shadow pages need to be maintained to keep track of which guest (i.e., Xen domain) is accessing which page, and being able to apply proper access control policies to avoid data leaking between virtual machines.

However, this solution creates a new challenge; instead of using the light-weight multi-threading, the approach will fall in the usual heavy multi-processing. To avoid this, it would be necessary to work on how I can apply the least privilege concept, reducing the amount of code I will clone for each thread to the minimum.

Other possible problems that could emerge could be the use of shared data among those threads or, in general, the management of those threads at the end of the execution. This could be a major limitation of the future work previously proposed on PRIVMUL, because it depends on the kind of job those threads are doing.

7.2.2 Automatic Detection Privilege Level Distribution

Relieving the developer from the effort of annotating code inside an application, the first step would be to study the C.G. of different applications with multi-user logic in such a way to find a common pattern. Then, I expect to create an automatic mechanism that makes educated guesses on the privilege separation. This idea could be designed and implemented in three different ways: on the source code of the application, directly on the binary or both.

What I expect to find out is that the C.G. of the application will match, more or less, the two schemes defined in Section 4.2. If the implementation is successful at the binary level, this will bring PRIVMUL also to work on proprietary software.

7.2.3 Interpreted Language Support

The last main future work I have planned is to look at interpreted languages and their virtual machines to see if the paradigm of PRIVMUL could be implemented also for those environments.

Interpreted languages do not suffer of memory corruption vulnerabilities due to the fact that programmers don't have to think how to create or delete objects. However, those lan-

guages are not perfect and programmers could introduce logical vulnerabilities that can bring an attacker to abuse of an application. It has been also demonstrated that vulnerabilities are presented by Brezinski in [2].

Exploring this family of languages could be very interesting because of their portability on different O.S.. It is also very fascinating to understand how much further I can apply PRIVMUL in the interpretation chain and see if I can avoid modifying the kernel of the O.S. using the virtual machine. If the kernel modification were no more necessary, the PRIVMUL approach would be usable also on other platforms rather than the only open source one.

7.2.4 Performance Improvement

Another improvement of PRIVMUL would be to modify the memory tracker to be faster than the actual implementation. A possible idea here would be to work on the whole page of memory instead of working on every single chunk of memory. This will allow to update the kernel side information only when a new page of memory is given to the application. This would reduce the number of the queries to the O.S., cutting down the overhead introduced by PRIVMUL.

Chapter 8

Conclusions

This thesis presents PRIVMUL, a novel and comprehensive approach to mitigate privilege escalation and data leak problems in applications that present multi-user logic.

Multi-user logic means that the application contains a certain number of functionalities which are divided into different profiles. The application could be used by one profile at a time, so only few functionalities are shown to the user. The problem in these applications is that the other functionalities are only hidden from the running profile using methods such as disabling a button or not showing the voice in the main menu. These methods are not effective when, in a running profile, an attacker is able to find a memory corruption vulnerability that gives him/her either the control of the application (brings the execution at a random location) or the possibility to read a random location in memory (leak of data that are created by other profiles in a previous session).

PRIVMUL avoids privilege escalation caused by memory corruption vulnerabilities. To this end, it protects (static) data and code according to simple policies that can be specified by the developer. The same concept is applied to dynamically allocated data thanks to the dynamic memory tracker proposed, which follows each allocation for every profile and protect them when needed.

PRIVMUL is a framework that provides new APIs to write multi-user logic applications in such a way that the O.S. is able to control and protect certain area of code and data when needed. Furthermore, these APIs could be used to instrument and adapt already existing applications with few modifications in case the source code is available. In this work, I in-

troduced two possible schemes of privilege distribution, the onion and the tree scheme. They must be kept in mind by developers when the application is written or adapted, in such a way that it would be possible for PRIVMUL to properly work.

PRIVMUL follows the application from the source code writing phase to the execution. It is composed by a static section (in which the application is partitioned) and a runtime section (where PRIVMUL actively interacts with the application). In the application it includes binary metadata that are generated during the building phase of the binary and then are brought to the O.S. kernel when the application is launched. In this way the O.S. is able to guarantee the integrity of the execution.

I implemented PRIVMUL in the GNU/Linux environment, keeping the kernel as the only trusted component to supervise the application execution. The Linux kernel is instrumented by the modified Glibc loader that I implemented, called *ld.so*, which, together with the usual segments structure, reads the sections information placed in the ELF header by the linker. The metadata is collected and stored during the compilation phase thanks to the three transformation passes that I wrote and a custom linker script generated by one of such passes. The pass generates, from scratch, a linker script for every analyzed binary. I wrote these passes for the LLVM/Clang compiler infrastructure.

During the execution, the application can change profile through a system call. The kernel receives the requests and compares the current profile with the new one. In case the application switches to higher privileges, an authentication and authorization routine starts. Authentication and authorization are implemented through a daemon that the kernel contacts via a netlink socket. This daemon uses PAM to authenticate the profile. The kernel starts the daemon and waits that the daemon authenticates the user. After the daemon that is running in a privileged mode has completed the authentication, it sends back the result to the kernel. If everything happens correctly, the kernel will change the permissions on the segments in memory and the new profile is unlocked.

PRIVMUL's overhead is significant on applications that use in-memory objects that are quickly created and destroyed. However, if I consider applications that fit the goals of PRIVMUL (i.e., large applications that make several interaction with the user and execute long functions), the overhead is acceptable (4x to 11x). Considering the effor required by the developer to secure a large application, the benefit brought forth by PRIVMUL is remarkable:

At the price of a simple annotation that specifies the access level of data or code, PRIVMUL takes care of all the rest. In other words, the balance between usability vs. security vs. performance overhead of PRIVMUL is significant.

Clearly, the current approach and its implementation have some limitations. On the approach side, PRIVMUL is requires the source code of the application, which may be an issue for legacy scenarios. Also, PRIVMUL requires O.S. modifications, which make PRIVMUL not immediately applicable to applications designed for closed-source systems. Similarly, PRIVMUL is not directly applicable to interpreted languages such as Python, Ruby or Perl. However, the generality of the approach does not exclude future extensions for interpreted languags. Last, multi-threading support is not implemented as it require an extensive amount of re-working (e.g., for handling shared memory and other IPC mechanisms).

In this thesis, I tackled the problems of privilege escalation and data leaking in applications with multi-user logic. The system security community have been studing these kinds of vulnerabilities at the O.S. level and proposed many solutions. However, I am the first to tackled these threats at the application level, proposing a solution that allows developer to write such fine-grained policies as thos of PRIVMUL.

This thesis was inspired by the new class of vulnerabilities presented by Mulliner et al. in [18] where the authors showed how, in many applications, the separation among users (usually the normal user and the administrator) was done naïvely by selectively hiding or displaying GUI elements (e.g., grayed-out buttons). They showed how the GUI could be misused to achieve privilege escalation in an application (avoiding access control schemes enforced). PRIVMUL gives an answer to this issue by providing a mechanism that can effective enforce ACLs on the application where the reference monitor is trusted in the O.S..

I believe that further work could be done in this direction to avoid this kind of threats at the application level. Indeed, even if the attack surface may be smaller than that of other threats, the consequences of privilege escalation are extremely serious. This is why studying these problems should be continued and ot provide flexible and usable solutions.

Bibliography

[1] James P. Anderson. *Computer Security Technology Planning Study*. Tech. rep. US Air Force, 1972. URL: http://csrc.nist.gov/publications/history/ande72.pdf (cit. on p. 7).

[2] Dominique Brezinski. "A Paranoid Perspective of an Interpreted Language". In: 2005. URL: https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-brezinski.pdf (cit. on p. 63).

[3] David Brumley and Dawn Song. "Privtrans: Automatically Partitioning Programs for Privilege Separation". In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 5–5. URL: http://dl.acm.org/citation.cfm?id=1251375.1251380 (cit. on pp. 3, 17).

[4] Bulba and Kil3r. "Bypassing StackGuard and StackShield". In: 2000. URL: http://www.phrack.org/issues/56/5.html (cit. on p. 21).

[5] Matt Conover and w00w00 Security Team. "w00w00 on Heap Overflows". In: 1999. URL: http://www.windowsecurity.com/uplarticle/1/heaptut.txt (cit. on p. 9).

[6] Crispin Cowan et al. *FormatGuard: Automatic Protection From printf Format String Vulnerabilities*. 2001 (cit. on p. 23).

[7] Crispin Cowan et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. 1998 (cit. on pp. 20, 21).

[8] Tyler Durden. "Bypassing PaX ASLR protection". In: 2002. URL: http://www.phrack.org/issues/59/9.html (cit. on p. 20).

[9] *Executable and Linkable Format (ELF)*. URL: http://www.skyfree.org/linux/references/ELF_Format.pdf (cit. on p. 13).

[10] Justin N. Ferguson. "Understanding the heap by breaking it". In: 2007. URL: https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf (cit. on p. 9).

[11] Mike Frantzen and Mike Shuey. "StackGhost: Hardware Facilitated Stack Protection". In: *In Proceedings of the 10th USENIX Security Symposium*. 2001, pp. 55–66 (cit. on p. 22).

[12] *FUSE: Filesystem in Userspace*. URL: `http://fuse.sourceforge.net` (cit. on p. 12).

[13] Kevin Kaichuan He. "Kernel Korner - Why and How to Use Netlink Socket". In: *Linux Journal* 7356 (2005) (cit. on p. 13).

[14] Douglas Kilpatrick. "Privman: A Library for Partitioning Applications". In: *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003, pp. 273–284. ISBN: 1-931971-11-0. URL: `http://www.usenix.org/events/usenix03/tech/freenix03/kilpatrick.html` (cit. on pp. 3, 17).

[15] *Linux Loadable Kernel Module HOWTO*. URL: `http://tldp.org/HOWTO/Module-HOWTO/index.html` (cit. on p. 11).

[16] *LLVM website*. URL: `http://llvm.org` (cit. on p. 10).

[17] "Malloc Des-Maleficatorum". In: 2005. URL: `http://phrack.org/issues/66/10.html` (cit. on p. 9).

[18] Collin Mulliner, William Robertson, and Engin Kirda. "Hidden GEMs: Automated Discovery of Access Control Vulnerabilities in Graphical User Interfaces". In: *IEEE Symposium on Security and Privacy (Oakland)*. San Jose, California, 2014 (cit. on pp. 6, 10, 66).

[19] Aleph One. "Smashing the stack for Fun and Profit". In: *Phrack* 49 (1996) (cit. on p. 8).

[20] Alexander Peslyak. *Getting around non-executable stack (and fix)*. Bugtraq mailing list. 1997. URL: `http://seclists.org/bugtraq/1997/Aug/63` (cit. on p. 23).

[21] Niels Provos. "Preventing Privilege Escalation". In: *In Proceedings of the 12th USENIX Security Symposium*. 2003, pp. 231–242 (cit. on pp. 3, 17, 18).

[22] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* (1975). URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=63665` (cit. on p. 2).

[23] *Secunia Database*. URL: `https://secunia.com/vulnerability-review/vulnerability_update_all.html` (cit. on p. 1).

[24] *SELinux Project Page*. URL: `http://selinuxproject.org/page/Main_Page` (cit. on p. 2).

[25] PaX Team. *NonExecutable Data Pages*. URL: `https://pax.grsecurity.net/docs/pageexec.txt` (cit. on p. 22).

[26] "The Malloc Maleficatorum - Glibc Malloc Exploitation Techniques". In: 2005. URL: `http://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt` (cit. on p. 9).

[27] *Tiobe - Programming languages statistics November 2014*. URL: www.tiobe.com/index.php/content/paperinfo/tpci/index.html (cit. on p. 7).

[28] Tim Twillman. *Exploit for proftpd 1.2.0pre6*. Bugtraq mailing list. 1999. URL: http://seclists.org/bugtraq/1999/Sep/328 (cit. on p. 9).

[29] Victor van der Veen et al. "Memory Errors: The Past, the Present, and the Future". In: *In the Proceedings of the 15th International Symposium on Research in Attacks Intrusions and Defenses (RAID)*. 2012 (cit. on pp. 7, 8).

[30] *What is Linux?* URL: https://www.kernel.org/linux.html (cit. on p. 11).

[31] *Winner of Pwnie Awards 2009*. URL: http://pwnies.com/archive/2009/winners/ (cit. on p. 1).

[32] *Winner of Pwnie Awards 2013*. URL: http://pwnies.com/archive/2013/winners/ (cit. on p. 1).

[33] Yongzheng Wu et al. "Automatically partition software into least privilege components using dynamic data dependency analysis". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 2013, pp. 323–333. DOI: 10.1109/ASE.2013.6693091 (cit. on p. 3).

[34] *ZFS On linux*. URL: http://www.zfsonlinux.org (cit. on p. 12).

December 18^{th}, 2014

Document typeset with LaTeX