Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Master Thesis in Computer Science

A Framework and Toolchain for Dropping Privileges

Author	Remo Schweizer
Main supervisor	Dr. Stephan Neuhaus

Date	May 14, 2018

Zürcher Hochschule
für Angewandte Wissenschaften

Life Sciences und
Facility Management

DECLARATION OF ORIGINALITY

Master's Thesis for the School of Life Sciences and Facility Management

By submitting this Master's thesis, the student attests of the fact that all the work included in the assignment is their own and was written without the help of a third party.

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree Courses at the Zurich University of Applied Sciences (dated 29 Januar 2008) and subject to the provisions for disciplinary action stipulated in the University regulations (Rahmenprüfungsordnung ZHAW (RPO)).

Town/City, Date: Signature:

…… ……..

The original signed and dated document (no copies) must be included in the appendix of the ZHAW version of all Master's theses submitted.

Zusammenfassung

Angriffe zur Ausweitung der Rechte auf einem System ist ein zunehmendes Problem aufgrund der steigenden Komplexität heutiger Anwendungen. Oftmals führen kaum erkennbare Fehler im Programmcode zu Schwachstellen mit enormen Auswirkungen. Insbesondere Sprachen wie C sind hierfür anfällig da diese aufgrund der verfügbaren Zeigerarithmetik die Ausführung unsicherer Operationen erlauben. Gelingt es einem Angreifer beispielsweise einen Speicherüberlauf zu provozieren, so kann ihm dieser Programmfehler die Möglichkeit verschaffen, beliebigen Code auszuführen. Um solche Attacken vorzubeugen oder zumindest deren Schadenswirkung zu minimieren, muss das prinzip der minimaler Rechtevergabe während des gesamten Entwicklungszyklus eingehalten werden.

Obschon diverse Tools und Funktionen zur Verfügung stehen um eine Anwendung zu Härten, so werden diese kaum genutzt. Unvollständige und teilweise falsche Dokumentationen sowie kaum vorhandene öffentliche Anwendungsbeispiele tragen ihren Teil dazu bei. Verstärkt wird das Ganze daurch, dass diese Methoden zur Aufteilung und Abstufung von Privilegie auf bestehende Softwarprojekte angewandt werden müssen.

In der vorliegenden Arbeit wird desshalb geprüft, wie die Standardfunktionen `LD_PRELOAD`, `Seccomp` und `PTrace` von Linux zur Steigerung der Sicherheit eingesetzt werden können. `LD_PRELOAD` erlaubt es die Ausführung von Bibliotheksfunktionen abzufangen und an einen Zweitprozess zu delegieren. `Seccomp` wird genutzt um innerhalb des Kernels über das Zugriffsrecht eines Systemaufrufs zu entscheiden. `PTrace` ermöglicht das Verhindern einer Ausführung eines potentiell schädlichen Systemaufrufs indem die Ausführung einer Anwendung überwacht und manipuliert wird.

Das Resultat dieser Arbeit ist ein hoch konfigurables Hilfsprogramm, dass automatisch die Privilegien zur Ausführung von Systemaufrufen trennt und widerruft. Eine speziell entwickelte Konfigurationssprache erlaubt es weiter Argumente der Systemaufrufe zu validieren und zu manipulieren ohne diese aufwendig in C implementieren zu müssen. Um die Integration in bestehende Anwendungen zu vereinfachen und ein Programm zu schützen reicht ein einziger Funktionsaufruf aus. Um den Integrationsprozess und die Möglichkeit zur Schwachstellenbekämpfung zu veranschaulichen, wird das Hilfsprogramm auf `Nginx`, einer der meist genutzten Webserver, angewandt.

Der Code zum Hilfsprogramm mit dem Namen `sec_seccomp_framework` steht öffentlich auf github zur Verfügung.
`https://github.com/deforation/sec_seccomp_framwork`

Abstract

Privilege escalation attacks are a serious problem for today's applications due to their increasing complexity. Often, barely recognizable programming errors can lead to vulnerabilities with immense consequences, especially in languages like C. They allow for unsafe operations, such as the manual addressing of memory regions using pointers. If an attacker can exploit a buffer overflow, he may find a way to inject arbitrary codes into an application and execute them, thus receiving its permissions. To mitigate such attacks and their consequences, the principle of least privileges must be established throughout the whole software architecture.

Albeit many functions and tools exist, that reinforce an application by dropping unused privileges and separating it into multiple parts with limited access rights, they are rarely used. Incomplete or incorrect documentation and the lack of public examples are often a hurdle, too. Particularly problematic is that they should be applied to the existing software solutions.

This thesis investigates the methods `LD_PRELOAD`, `Seccomp` and `PTrace` offered by the Linux kernel and their use for securing software solutions. `LD_PRELOAD` wraps library functions to delegate their execution to a request broker. `Seccomp` performs custom system call permission checks at the kernel level. `PTrace` can be used to disallow potentially vulnerable system calls by monitoring and manipulating the execution flow of an application.

The solution proposed in this work to implement the concept of least privileges is a highly configurable framework which takes care of dropping privileges, data validation, data sanitization and the separation of privileges at the system call level. An extensive language is created to configure the framework without the need to manually code parameter validation and sanitization checks in C. To secure an existing or new application, a single function call to the framework is sufficient. An example integration of the framework into one of the most used web servers called `Nginx` demonstrates the work flow and its capability to mitigate existing and potential vulnerabilities.

The framework called `sec_seccomp_framework` is publicly available on GitHub.
`https://github.com/deforation/sec_seccomp_framwork`.

Management Summary

This work proposes a novel framework to integrate the concept of least privileges in a few steps into new and existing applications, which developed in C. An extensive configuration language supports potential users to build rules for data validation and sanitization on the system call level. Access to these privileged function calls can be restricted within the kernel and user space of Linux. To ensure seamless portability of the framework, only functions provided by the official Linux kernel are used. A special focus is placed on the simplicity of integrating the framework into software projects. The invocation of a single function call is sufficient to protect an application.

The implementation of this framework requires solid understanding of the field of privilege escalation attack prevention. Niels Provos et al. conclude that the separation of privileges is an absolute necessity [1]. Based on this knowledge, scientific research papers with the focus on privilege separation were collected. Four publications gained special interest because of their focus on automating this process for generic applications.

Privman delegates the execution of privileged actions to a request broker by providing a range of wrapper methods for commonly used functions provided by the standard C library. The replacement of the original function calls by the wrapper version must be performed manually [2].

Privtrans also delegates the execution of privileged actions to a single request broker. The invocation of privileged functions occurs automatically through annotations in the source code placed manually by a developer with expert knowledge of the application [3].

ProgramCutter extends the previously proposed solutions and eliminates the need for expert knowledge. Execution traces are used to automatically cut an application into privileged and unprivileged parts. This method assumes that the privileged parts are small and that the generated pieces are free of vulnerabilities [4].

SPL|T further removes the need for manual intervention by extending dynamic analysis (execution traces) with static analysis (source files) [5]. In addition, privileged parts gain the ability to interact with each other. Data validation and sanitization must still be performed manually.

Contrary to the framework proposed in this work, all the mentioned solutions lack one crucial information, which is how and to what extent privileges can be dropped and are dropped. This not only targets the unprivileged part of a split application, but also privileged parts should be restricted in their actions. As a result, methods were examined how privileges on Linux-based operating systems can be dropped. To this extend the methods `LD_PRELOAD`, `Seccomp` and `PTrace` are found to be most suitable. Considering those, software architectures were designed and analyzed in their potential to drop and split privileges in a secure and comprehensive manner.

The solution found is a combination of `Seccomp` and `PTrace`. `Seccomp` is used to define access rights to system calls at the kernel level. Validation checks which are out of scope for `Seccomp` are delegated to the request broker using `PTrace`. `PTrace` further allows emulating system calls and modifying their arguments.

We demonstrate the integration and capabilities of the proposed framework by applying it to, `Nginx`, one of the most used web servers. In particular, we show how known vulnerabilities can be weakened and attack scenarios blocked.

Preface

First and foremost, I want to thank my supervisor, Dr. Stephan Neuhaus for his excellent work supporting this thesis. He was also the one coming up with the idea to build a tooling for privilege dropping and separation. Because of to my interests in the field of IT-security and software development in general, it was natural for me to explore and give shape to this idea. Not only am I pleased with the outcome of this work, it also gave me a wonderful opportunity for deepening my knowledge on Linux-based operating systems. I would also like to thank my family for giving me the chance to work full time on this project. Special thanks go to my brother, Dr. Marco Schweizer and his wife, who actively supported me with advice on a wide range of different topics. Lastly, I want to thank my dog, Mocha. She entered my life as a young puppy when my work was nearing its end. She taught me a sense of responsibility and a value of regaining energy through a break as the final steps of the work were becoming tougher and more challenging.

Contents

1 Introduction

A data breach and infected computer systems caused by the application of inadequate security mechanisms to any public and private software are no rarity. Successful exploitations of vulnerabilities can bring about business image damage or the loss of sensitive data and documents. Equifax, a consumer credit report agency which aggregates data of over 800 million individuals and 88 million businesses worldwide, is one of them. During the data breach in 2017, attackers were able to steal personal information such as social security numbers and credit card credentials of more than 143 million US citizens by exploiting the vulnerability CVE-2017-5638 of apache struts [6, 7]. If the file upload module is handled incorrectly, commands to remotely execute malicious codes could be crafted because the error handling has not been implemented sufficiently. Such an implementation mistake cannot always be prevented, especially within large and complex software architectures. The central issue provoked by this incident is why the upload module had permission for accessing confidential files and execute custom code. According to the SANS Institute, executing applications and modules with unnecessary privileges is the eleventh most dangerous and common software error [8]. The possible explanations for the magnitude of this type of problems can be found in the way today's computer systems are built and used. Even though operating systems offer the capability to run multiple applications with different privileges simultaneously, the functionalities are used inadequately. Applications are often run with root privileges giving them full control over the system, or permissions are disproportionately separated among users. In both cases, cyber-criminals may gain access to rights that they were not allowed to receive. This procedure is, in general, summarized under the term, privilege escalation.

Common concepts to mitigate privilege escalation attacks are privilege dropping and privilege separation [1]. Privilege dropping describes the process of revoking permissions to certain resources when they are no longer needed. Privilege separation, on the other hand, describes a concept where permissions are split among different execution entities (processes). Although current operating systems, especially Linux, provide various tools to integrate these concepts into applications, they are often difficult to understand or use due to their insufficient documentation. The main goal of this work is, therefore, to build a framework for integrating these two aspects into new and existing applications. A particular focus is placed on simplicity, which gives even inexperienced developers the ability to increase the security of their applications. The target programming language of the framework will be C due to its extensive distribution and acceptance within the Linux community. Another reason for C is that unsafe operations have great potential for introducing errors into an application.

A widely used approach to implement privilege dropping and separation was introduced in the early days of Linux. The functions, `setuid` and `getuid`, allow an application to change its affiliation with a certain user and group at runtime. This allows permissions to be modified dynamically because Linux restricts access to system resources based on user and group IDs. Despite the sophisticated access control mechanism, the methods come with several drawbacks due to their historical background. H. Chen et al. have taken a deeper look into these functions [9]. They have come to the conclusion that the behavior of the functions has altered over time and that because of this, the documentation of these functions has become incomplete and sometimes even wrong. The complexity has further increased since there exist three versions of the user and group ID. The real user ID identifies the owner of the process. The effective user ID is used in access control mechanisms, and the saved user ID is a backup of the original user ID in case privileges have to be restored. Nonetheless, a more serious drawback is that for every permission configuration, a user account must be created.

Besides `setuid` and `getuid`, Linux offers additional tools to drop and separate privileges. `PTrace` described in Sec. 2.5 is able to observe and manipulate the execution flow of an arbitrary process,

thus indirectly allowing for options to hinder the execution of potentially vulnerable system calls. `LD_PRELOAD` described in Sec. 2.3 can be used to wrap functions of dynamically linked libraries offering the opportunity to redirect their execution to a request broker with more privileges. `Seccomp`, described in Sec. 2.4, offers the ability to load a custom filter program into the kernel to permanently revoke access to certain system calls.

In any case, the concepts of privilege dropping and separating permissions are usually manually integrated into new and existing applications. Some researchers, therefore, evaluated different methods for automating this process. For example, Derek G. et al. present a concept where a process is disaggregated at the level of dynamically loaded libraries [10]. Each library will be executed in its own virtual machine, thus protecting its data from the access by potential malicious functions of other untrusted libraries. With this approach, privilege escalation attacks are still a serious problem because system calls are executed on the host system to which the main application has full access. A completely different approach has been developed by Juan W. et al. and their feasible solution is Arbiter [11]. The primary focus of Arbeiter is to achieve a fine privilege separation in multithreaded applications. To accomplish this task, a part of the memory management system of the kernel is rewritten to support permission bits. These bits can prevent mutual data access among the threads running in the same address space. Arbiter applies the concept of least privileges to shared data objects. However, this contradicts the objective of minimizing the privileges of the whole application. Portability also suffers due to the use of a custom Linux kernel. Ira Ray J. et al. present a different perspective on their approach to build a new concept for dropping privileges [12]. Adding security policies at the Application Binary Interface (ABI) level helps to define semantic relationships between code and data. Like the approach of Derek G. et al., the concept solely focuses on restricting access to data. The access to system calls and its risk for privilege escalation attacks remain untouched. Privman achieves privilege separation by running a privileged child process as a request broker [2]. Alternative versions for common used functions such as `priv_fopen` for `fopen` can then be used to outsource the execution to this specific request broker. Using this approach, an attacker gaining the privileges to execute arbitrary codes can just invoke the `open` system call directly in order to open a file. In this way, the security measures are circumvented. Unlike Privman, in which every function call must be manually replaced by a Privman alternative, Privtrans extends the automation process [3]. With Privtrans, a developer can simply achieve privilege separation by adding annotations to the source code. The framework then automatically executes these privileged actions in a separate child process. Even though it offers more possibilities and a simpler interface than Privman does, it is prone to the same attack vector as Privman. ProgramCutter takes a similar approach to Privtrans but enhances the concept in such a way that no expert knowledge about the software application is required [4]. The tool automatically detects and writes privileged acions at the system call level to use special wrapper functions provided by ProgramCutter. This approach assumes that the privileged sections that it generates are free from vulnerabilities and that the execution traces used to cut the application are complete. The last point is rarely the case, especially in large and complex software solutions. SPL|T handles this problem and circumvents it by adding a configuration file and a statistical code analysis [5]. It provides a further solution to the problem when privileged parts call functions mutually. This particular problem is resolved by creating IPC-based communication gateways between the privileged parts of the application. Also in this approach, data validation and sanitization for the privileged functions must be implemented manually in order to prevent an attacker from misusing them through a loophole in the unprivileged process.

The common problem of the previously described methods is the lack of information on how to drop unnecessary privileges so as to fulfill the concept of least privileges. They primarily focus on the granularity and method at which an application is split into privileged and unprivileged components. Furthermore, in all cases, data validation and sanitization must be implemented manually within the generated wrapper functions.

To solve these problems a novel highly configurable framework that takes care of dropping privileges, data validation, data sanitization and the separation of privileges is proposed. In addition, it completely relinquishes the need for time-consuming marshaling and unmarshaling operations to pass function arguments between privileged and unprivileged entities. Meanwhile, there is also an assumption that system calls represent the privileged operations. This is valid because invoking system calls is a prerequisite for accessing any kind of system resource. The core element of the framework is a symbiotic combination of `Seccomp` and `PTrace`. On top of that, an extensive language is created to configure the solution without the need to manually code parameter validation and sanitization checks in C. To integrate the tool into the existing applications, a simple function call is sufficient without having to define annotations within the source code or to replace function calls manually.

An overview of privilege separation and dropping in general is given in Chap. 2. In particular the corresponding methods - `LD_PRELOAD`, `Seccomp` and `PTrace` - and their functions provided with the Linux kernel are introduced in Sec. 2.4 and Sec. 2.5. Based on these methods various software architectures that allow for full control over the system calls, irrespective of the location of their invocation, are presented in Sec. 2.6. The most promising architecture related to this task and other important criteria such as multithreading and performance is then used as a foundation for the proposed framework.

Chap. 3 develops the final implementation of the novel framework. The relation of the framework to `Seccomp` and `PTrace` and their mutual interaction is described in Sec. 3.1. In addition, various peculiarities of these methods encountered throughout the implementation process are reviewed. This especially includes their behaviour in threads and forks. In Sec. 3.2 the limitations of the proposed framework are considered. Sec. 3.3 and Sec. 3.4 show how access rules to system calls, their argument validation and sanitization rules can be configured.

Chap. 4 shows the integration of the framework into the popular web server `Nginx`. Its effectiveness to mitigate the potential damage to a system and to reduce the attack surface is demonstrated by applying it to different attack scenarios and known vulnerabilities.

2 Methods

In a preliminary evaluation phase of this work, techniques for dropping and separating privileges on a Linux-based operating system were investigated. The following chapter presents various methods for dealing with privilege schemes. First, a brief summary of the concept of privilege separation is stated. Subsequently, the mechanisms for privilege dropping are introduced. The subsequent sections, namely `LD_PRELOAD`, `Seccomp` and `PTrace` describe the actual methods considered for use in this work. The last part presents four different architecture types that combine these methods. Each method is briefly described and compared, taking into account key factors such as the ability to intercept system calls, the limitations of the function space and the overall performance and security.

2.1 Privilege Separation

Privilege separation describes the concept of splitting privileges across multiple independent execution entities communicating through an I/O channel like sockets. The goal, therefore, is to give each subprocess of an application minimal privileges to fulfill its intended task. For instance, if an application processing user input has no rights to read and write files, it would outsource the very job to a secondary process, called request broker, that possesses the necessary rights. Even if the main application gets hijacked due to bugs or insufficient input validation, the system cannot be harmed unless the attacker, who has the necessary rights, can craft malicious requests to the request broker. Therefore, it is important to minimize the communication interface between the processes in order to reduce the likelyhood to introduce new loopholes. In addition, each process should examine strictly weather the received requests belong to the normal application behavior and that they are equipped with a well-defined structure.

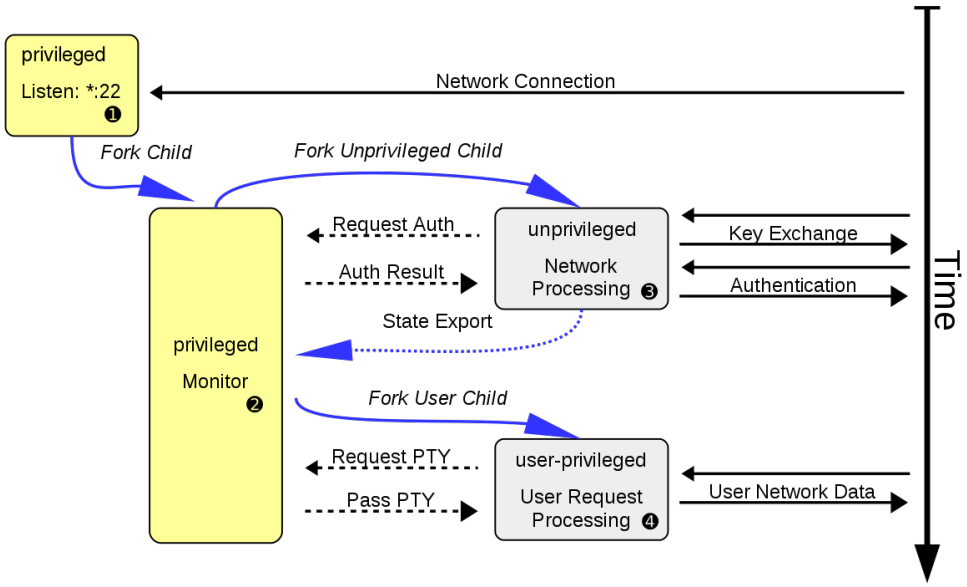

Fig. 1: Sequence Diagram for `OpenSSH` Privilege Separation [13].

An example using privilege separation is the `OpenSSH` project shown in Fig. 1. This remote login tool, based on the SSH protocol, has been extended by the privilege separation mechanism in 2002. The aim was to separate the pre-authentication logic to prevent the remote compromise before and after authentication. A root process called monitor creates an unprivileged child

process called slave for all the communication attempts, which then handles the authentication logic. All the accumulated results are then sent to the root process, which decides weather the access will be granted. In the early versions of `OpenSSH`, an attacker was able to craft messages that would lead to an escalation of privileges. This risk is significantly reduced in the modern architecture: If the slave process wrongly determines that the other endpoint is authenticated, it will be corrected by the monitor.

`qmail` is another popular example of privilege separation. Compared to `OpenSSH`, the segregation level of this mail transfer agent is much more fine-grained. The whole application consists of more than 24 separate Unix programs [14]. To fully take advantage of the security mechanisms, up to seven system users and groups must be created and configured. To receive a mail from the smtp gateway and its transfer to a local mailbox, at least five privilege boundaries are traversed. First, a mail is received by the module qmail-smtp running as qmaild user. It is then stored in a queue managed by the process qmail-queue running as qmailq user. qmail-send running as qmails user scans the queue and transfers it further to qmail-lspawn running as root user. The last process in the execution flow is then qmail-local running as a local user before it is sent to the mailbox. The advantage of such fine granularity is without question its security aspect. The small and independent modules drastically reduce the attack surface and the difficulty for a developer to understand a single module. On the other hand, understanding the whole program and connections between the modules becomes more complex and complicated to maintain. The installation process also becomes circuitous. For example, the necessary system user accounts and groups prevent Linux distributions from offering a ready-to-use binary.

The concept of privilege separation has many advantages, but it only offers a way of how privileges for a single process can be reduced. It does not describe, though, how commands can be intercepted at the core level. In Linux terminology, this is dealt with *"system calls"*. If the goal is to provide an easy-to-use framework to reduce the risk of privilege escalation for any Linux-based application, it is necessary to intercept the core instructions that can be used for malicious actions. To solve this problem, combining the concept of privilege dropping and privilege separation is mandatory.

2.2 Privilege Dropping

Applications normally possess an excess of privileges that they do not require. For instance, some applications have rights to read and write to non-essential partitions and folders. This becomes a security risk when they process variable user input or receive and parse data from an I/O-interface: if an application contains bugs or wrongly verifies the user input, it is more probable that the underlying system is harmed by the specific queries crafted by an attacker. Confidential information might end up leaking, or the system itself might get infiltrated with harmful code.

To reduce the risk of an application being taken over and to gain root privileges, Linux-based operating systems have integrated the functions `setuid` and `getuid`. These functions allow developers to change a set of user and group identifiers to modify the privilege set of an application. As a result, an application can read sensitive data at the initialization phase and remove the privileges afterwards. In the early stages of Linux, an application was unable to drop privileges temporarily. To solve this problem, modern Unix and Linux systems have introduced a set of different identifiers to handle temporary modifications and a finer granulation of privileges. An application no longer decides between root and non-root but instead, the privileges for specific resources can be defined.

Although `setuid` and `getuid`-based approaches offer a large variety of methods for dropping privileges temporarily or permanently, they are often used improperly: if an attempt to change

the user ID defining the permissions is sent, the application can keep running with the original privileges in case the system was unable to change the specific ID. Therefore, if the return value of the functions is not verified, applications can run with unintended rights. The failure of proper use can be partially attributed to the different stages of Unix and Linux, as the working principle of the functions changed and has become more vaguely documented. For instance, Hao Chen and David Wagner analyzed the methods in detail, illustrating their complexity and use case [9].

Another important aspect of dropping privileges is the duration. Temporary privilege dropping may be helpful if done before parsing critical user input and ending it afterward. In certain situations, this can also lead to severe security risks if an attacker can reset the privileges to a higher level. This can never be prevented entirely because the application itself has the right to perform this action. As a result, privileges should always be dropped permanently with no possibility to re-enable them.

In 2005, a novel method called `Seccomp` was implemented for Linux-based operating systems. This security mechanism enhances the concept of dropping privileges and offers a wide variety of options for customization. The concept is so fundamental, that it is widely used in larger projects such as Google Chrome and Firefox. `Seccomp` basically allows an application to perform checks for system calls at the kernel level. It is therefore beyond question, under consideration of a detailed method comparison in Sec. 2.6.5, that this approach plays a significant role in the final implemented framework of this thesis.

2.3 Concept of `LD_PRELOAD`

`LD_PRELOAD` describes a way to intercept existing function calls from dynamically loaded libraries. It therefore, provides the opportunity to modify wrapper functions to system calls defined in glibc, uclibc and others. This only requires defining the shared library with the modified function calls and running the application of interest. Code changes to the application are hence not needed, see for instance the example source shown in Algo. 1. Linux has two `open` system calls, `open` and `open64` (the 64-bit version of `open`), for which a wrapper version in glibc exists. An application using the wrapper function in algorithm, Algo. 1, would write *"open called"* to the standard output each time an attempt is made to call `open`. When an application intends to run a function called `open`, the linker searches for the symbol (function call) in a dynamically linked library. Once that symbol is found (function definition), it is executed. It may be now possible that multiple definitions exist for the function call `open`. In this case, the linker takes the first one matching the argument types of the function call. `LD_PRELOAD` now adjusts the linker such that the symbol will be first searched within the defined library. As a result, the defined wrapper function will be called first. The major drawback of this method is that system calls cannot be intercepted because they are called differently. To run a system call, the application sends an interrupt to the kernel specifying the system call by a numerical representation within one of the CPU registers.

In conclusion, `LD_PRELOAD` defines a simple way to intercept function calls in dynamically loaded libraries. On the other hand, this method has several drawbacks. There is no way to intercept system calls at the core level. In addition, even if all functions of glibc would be wrapped using this method, any third-party library could directly trigger the kernel without using the standard C library. It would therefore be necessary to scan all the linked files of an application if they perform a system call. If this is the case, a wrapper function would have to be generated for it with appropriate logic to perform sanity checks on the system call parameters.

```
1  #define _GNU_SOURCE
   #include <dlfcn.h>
3  #include <stdio.h>

5  typedef int (*orig_open_function_type)(const char *pathname, int flags);

7  int open(const char *pathname, int flags, ...)
   {
9    printf("open called");

11   orig_open_function_type orig_open;
     orig_open = (orig_open_function_type)dlsym(RTLD_NEXT,"open");
13   return orig_open(pathname,flags);
   }
15
   int open64(const char *pathname, int flags, ...)
17 {
     printf("open called");
19
     orig_open_function_type orig_open;
21   orig_open = (orig_open_function_type)dlsym(RTLD_NEXT,"open64");
     return orig_open(pathname,flags);
23 }
```

Algo. 1: `LD_PRELOAD` example for the open system call.

2.4 Concept of `Seccomp`

`Seccomp` was introduced with Linux 2.6.12. To fully drop the privileges for all system calls except `read`, `write`, `exit` and `sigreturn` an application just must write *"1"* to *"/proc/PID/seccomp"*. This interface has undergone several changes in 2007, 2012 and 2014. First, the command `prctl` was introduced to initialize `Seccomp`. Second, the Berkeley packet filter (BPF) language was introduced, which allows a custom behavior to be defined for each system call with consideration of its arguments [15]. In 2014, the Linux kernel version 3.17 was released, further simplifying `Seccomp` by adding a function of the same name instead of extending the parameter set of its complex predecessor `prctl`.

One reason for the long development process of `Seccomp` to reach its advanced state is related to it starting from a niche business idea to securely buy and sell central processing unit (CPU) power from and to third-parties [16]. This business case was also the reason for allowing only four system calls in the early phase. `Seccomp` then remained most likely unused in the Linux kernel until Google used it to sandbox plugins for Chrome. During their research, they stumbled on `Seccomp` and extended it with custom filters functionality described subsequently.

2.4.1 Berkeley Packet Filters

The customization of validation checks of system calls is achieved by applying Berkeley packet filters already used on the raw interface of a network's data link layer to a well-defined data structure for system calls. Algo. 2 shows an example of a `Seccomp`-based Berkeley packet filter program. At the beginning of a filter program, the architecture must be checked. It assures the correctness of the numerical representation of the system calls. If this check is ignored, the program execution could result in unexpected behavior. On an x64 architecture, the open system call could have the numerical representation of *"1"*. On an ARM-based system, on the other hand, it could be resolved to *"2"*. Therefore, the same rule could have different outcomes. Subsequently, different rule checks take place from line 5-42. In the end, the possible actions that can be returned by the filter program are noted.

```
1  // Architecture check (important for correct system call number resolve)
   BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, arch)),
3  BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 0, 20),

5  // Check rule 1 (system call exit_group is allowed)
   BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr)),
7  BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_exit_group, 19, 0),

9  // Check rule 2 (system call exit is allowed)
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_exit, 18, 0),
11
   // Check rule 3 (system call read is allowed)
13 BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_read, 17, 0),

15 // Check rule 4 (system call write is allowed)
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_write, 16, 0),
17
   // Check rule 5 (system call socket is allowed)
19 // for AF_LOCAL connections based on a STREAM
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_socket, 0, 4),
21 BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[0]))),
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, AF_LOCAL, 0, 2),
23 BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[1]))),
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, SOCK_STREAM, 11, 0),
25
   // Check rule 6 (system call socket is allowed)
27 // for AF_UNIX connections based on a STREAM
   BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[0]))),
29 BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, AF_UNIX, 0, 2),
   BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[1]))),
31 BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, SOCK_STREAM, 7, 0),

33 // Check rule 7 (system call setrlimit should fail)
   // if we try to change the CPU limit
35 BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr)),
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_setrlimit, 0, 2),
37 BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[0]))),
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, RLIMIT_CPU, 4, 0),
39
   // Check rule 8 (system call open will be redirected to a tracer)
41 BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr)),
   BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 3, 0),
43
   // Action executions according to the rule
45 BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
   BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
47 BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO | (327681 & SECCOMP_RET_DATA)),
   BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_TRACE | (16 & SECCOMP_RET_DATA))
```

Algo. 2: `Seccomp` Berkeley packet filter example.

Each check basically consists of three steps. First, the value to be checked is loaded into the accumulator. This action is done by using the command `BPF_STMT` as shown in the lines 6, 21, 23, 28, 30, 35, 37 and 41. Second, a conditional jump takes place by comparison of a custom value against the accumulator. Depending on the outcome, a jump takes place instructing the kernel to skip a specified number of lines within the BPF-program. In terms of line 7, the system call number is checked against the numeric representation of the system call `exit_group`. If it succeeds, the interpreter jumps to line 46. As the third step, the program returns the action *`SECCOMP_RET_ALLOW`*, instructing the kernel to execute the system call. However, if the check fails, no line is skipped, but rather the next line of the program is executed. Checking the value of system call arguments can be done in the same way as checking the numeric representation

of the system call. Line 21 shows an example in which the first argument is loaded into the accumulator. The next line then compares it with the value of `AF_LOCAL`. Besides the possibility to allow a system call, `Seccomp` offers five additional return values:

- *Kill*:
  terminating an application.
- *Trap*:
  sending a *SIGSYS* signal to the application without execution of the system call.
- *Errno*:
  letting the execution fail and returning an error number.
- *Log*:
  executing the system call and noting its execution within the system log file.

`Seccomp` BPF-programs are subject to several restrictions. Compared to its counterpart version for network packets, only one register/accumulator is supported. Hence, there is no possibility to load the system call number and the arguments into separate registers to check them there. Furthermore, there is no way to perform jumps with a negative offset. Therefore, loops are not supported and the program will always end. The severest limitation is that it is not possible to inspect data behind a pointer. `Seccomp` allows the address of the pointer to be checked but not the target to which it refers. This can only be achieved by either calling a trap function as a system call result or by notifying a tracer.

2.4.2 Trap Functions

`Seccomp` allows a trap function to be called before a system call is executed. This is done by returning the value *SECCOMP_RET_TRAP* within the filter program. As a result, the kernel sends a *SIGSYS* signal which can be intercepted by the application. In the signal handler, it is possible to get information about the system call and its parameters. In contrast to `Seccomp`, the signal handler allows the application to inspect pointer-based arguments. The example code in Algo. 3 shows how such a trap function may look like. In the beginning, the trap function must be initialized, using `sigaction` in line 62 to set the signal handler. Thereafter, all trap function calls will call the function `seccomp_trap` in line 36. Within the trap function, it is important to check the `si_code` to ensure that the signal was triggered by `Seccomp`. If this is the case, the system call arguments can be retrieved from the registers.

Due to the nature of signal handlers on Linux-based operating systems, the allowed actions in the signal handlers are limited. All functions used within the handler have to be async-signal-safe. This is the case when a function can be interrupted multiple times within its own execution without accessing, generating or working on inconsistent data. This dramatically reduces the number of allowed system calls. In addition, the `Seccomp` trap function prevents the initially-called system call from execution, causing it to fail. On the other hand, it is possible to modify the return value of the system call so zhat it can be emulated. However, it is not possible to emulate all system calls because only a fraction is async-signal-safe according to POSIX.1 [17]. Inspecting the arguments of the system call `mmap`, which is used, e.g., to map files or devices into memory, may be done, but the emulation is rather difficult and insecure due to the reasons discussed below.

Emulation

Suppose that an inspection of `mmap`'s arguments shows that the system call should, in fact, be executed. Since the `Seccomp` BPF-program has already been executed, the `mmap` call has been abandoned, and the signal handler has been called instead. There is, therefore, no way to simply resume the `mmap` call. One could instead try to emulate `mmap`, but the effects of `mmap` are unique

to it, and so there is no way of doing what `mmap` does using other system calls. In fact, the only solution would be to call `mmap` from the signal handler, but that also fails because `mmap` is not guaranteed to be async-signal-safe. And even if it were, all this would cause is another run through the BPF-program and another call to the signal handler, creating an infinite loop.

```
#define SECCOMP_REG(_ctx, _reg)    ((_ctx)->uc_mcontext.gregs[(_reg)])
#define SECCOMP_RESULT(_ctx)       SECCOMP_REG(_ctx, REG_RAX)
#define SECCOMP_SYSCALL(_ctx)      SECCOMP_REG(_ctx, REG_RAX)
#define SECCOMP_PARM1(_ctx)        SECCOMP_REG(_ctx, REG_RDI)
#define SECCOMP_PARM2(_ctx)        SECCOMP_REG(_ctx, REG_RSI)
#define SECCOMP_PARM3(_ctx)        SECCOMP_REG(_ctx, REG_RDX)
#define SECCOMP_PARM4(_ctx)        SECCOMP_REG(_ctx, REG_R10)
#define SECCOMP_PARM5(_ctx)        SECCOMP_REG(_ctx, REG_R8)
#define SECCOMP_PARM6(_ctx)        SECCOMP_REG(_ctx, REG_R9)

// System call emulation multiplexer routine
greg_t emulate_syscall(int syscall, greg_t par1, greg_t par2, greg_t par3,
                       greg_t par4, greg_t par5, greg_t par6){
  switch(syscall){
    case SYS_open:
      return (greg_t)sec_open((char*) par1, (int) par2, (mode_t) par3);
      break;
    case SYS_chdir:
      return (greg_t)sec_chdir((char*) par1);
      break;
    case SYS_getcwd:
      return (greg_t)sec_getcwd((char*) par1, (size_t) par2);
      break;
    default:
      printf("INVOKED NON CHECKED SYSTEM CALL\n");
      return (greg_t)-1;
  }
}

// Catch violations so we see, which system call caused the problems
static void seccomp_trap(int sig, siginfo_t* si, void* void_context)
{
  if (si->si_code != SYS_SECCOMP)
    return;

  int old_errno = errno;
  ucontext_t* ctx = (ucontext_t*)void_context;
  SECCOMP_RESULT(ctx) = (greg_t)emulate_syscall(
                SECCOMP_SYSCALL(ctx), SECCOMP_PARM1(ctx), SECCOMP_PARM2(ctx),
                SECCOMP_PARM3(ctx), SECCOMP_PARM4(ctx), SECCOMP_PARM5(ctx),
                SECCOMP_PARM6(ctx));

  errno = old_errno;
}

// Initializes the trap function for seccomp
static void init_seccomp_trap(){
  struct sigaction sa = {
    .sa_sigaction = seccomp_trap, .sa_flags = SA_SIGINFO | SA_NODEFER
  };
  if (sigaction(SIGSYS, &sa, NULL) == -1){
    printf("Error in initializing the seccomp signal handler -> [%s]\n",
    strerror(errno));
  }
}
```

Algo. 3: `Seccomp` trap function handling example.

Broker

Assuming that an application intends to inspect the arguments of the system call `open` and it decides to allow for access to a certain file. In this scenario, basically, the same problem as described in the previous paragraph occurs. However, there are two ways to circumvent this problem. First, the trap function could run a similar system call such as `openat` to emulate it. However, this is nonsensical because all the parameter checks would be obsolete if `openat` would be executed without the same checks. It is, therefore, mandatory to assume that `openat` will be checked under the same rules and that its emulation is also required, which ultimately leads to the endless emulation loop problem. The second solution is to introduce a broker service. The idea behind the broker service is that the main application has no right to call `open`. The broker, on the other hand, has the necessary rights to do this. This scheme is called privilege separation as described in Sec. 2.1. In terms of `open`, the broker would perform the necessary checks and call the function on behalf of the main application. As a result, the broker would pass the resulting file descriptor to the main application.

Multithreading

The described broker service may be a good way to support emulation for some of the system calls. However, the broker is not able to execute every system call on behalf of processes, which makes it only a partial solution for the overall problem. More problematic are applications that incorporate multithreading. This is because all threads and the main application use the same signal handler. This requires that either each thread create its own socket connection to the broker or that the thread ID is needed in any transmission for identification in case all threads use the same socket. The main reason is that the specific trap function of the application is executed by the calling thread itself. Moreover, the trap function call can run concurrently, which explains one reason why the signal handler must be async-signal-safe and functions like malloc are not allowed due to their use of static data structures. Thus, a solution working with only one socket connection will take into account that if multiple threads run the handler at the same time, checks must be performed to ensure that each thread receives the return value that was meant for it. Another possibility one can come up with is to set a lock around the trap function, so it is executed sequentially. This will not work either, because the required system calls do not have the property of beeing signal-safe.

2.5 Concept of `PTrace`

`PTrace` was first introduced in 1975 with the sixth version of Unix. It allows a process *"tracer"* to observe and control the execution of another process, which is a so-called tracee. Typically, this tool is used to implement debuggers or application analysis tools to inspect not only system calls but also the overall internal state of the application and its registers.
As mentioned in Sec. 2.4, `Seccomp` supports the return type *`SECCOMP_RET_TRACE`* allowing a tracer to be notified right before the execution of a system call. It gives the tracer the possibility to perform the following actions:

- Inspect/Modify the arguments of a system call.
- inspect/Modify the return value of a system call.
- Resume the execution of a system call.
- Prevent the execution of a system call.
- Change the system call into another one.
- Terminate the application (tracee).

There are several advantages of `PTrace` over the trap function. The most crucial benefit is the ability to allow the execution of a system call after the inspection or manipulation of one or

more of its arguments. Furthermore, `PTrace` offers built-in tools to deal with multithreaded applications, even if they are not the most performant ones. Instead of duplicating the execution code of the tracer, all threads are paused until the tracer has finished its job. This ensures that no data inconsistencies can occur while the application is inspected.

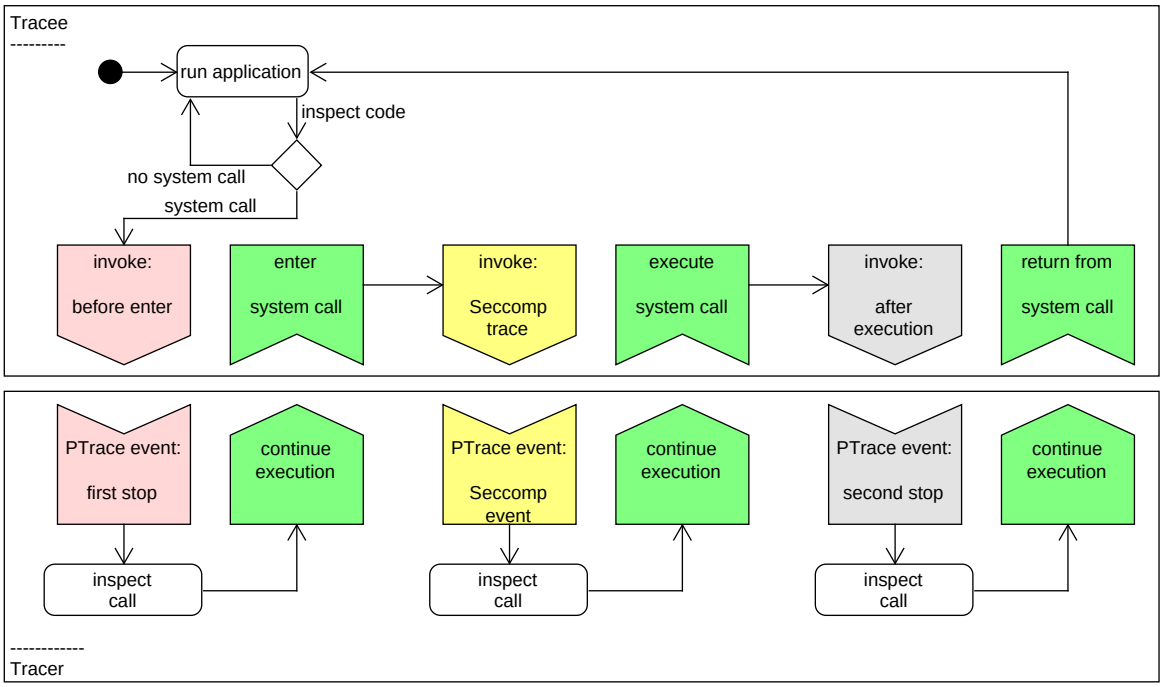

Fig. 2: System call execution path using `PTrace` with `Seccomp`.

Combining the capabilities of `PTrace` with `Seccomp` gives rise to the possibility of inspecting a system call in three different states as shown in Fig. 2. First, the built-in mechanisms of `PTrace` allow inspecting a system call right before it is executed. During this event, the system call number can be changed so that a completely different system call is executed. The tracee starts the execution once the tracer gives its permission. Now, if `Seccomp` is used with a rule defined to inform the tracer, the kernel sends an event-based on `Seccomp` to inform the tracer again. This is the optimal stage to check the arguments and modify them if necessary. Changing the system call to execute is still possible. The kernel resumes its execution when the tracer gives its permission. After the execution of the system call, the tracer gets another event notifying it of its termination. At this point, the manipulation of system call arguments is meaningless unless it is related to a parameter used as a return value. A possible example is the data within the buffer of the `read` system call, which allows data from a given file descriptor to be read. At this stage, the system call cannot be aborted anymore because the system call may have already changed the internal state of the system. After the permission of the tracer, the application continues with the execution of code instructions until the next attempt to execute a system call is made.

There are different ways of integrating `PTrace` in the privilege dropping and separation approach.

2.5.1 Changes on the Runtime Behaviour

One way to give a developer an easy way to write emulation functions for system calls with the ability to check system call arguments would be to write a wrapper version of the system call within the same application by using exactly the same arguments. Fig. 3 shows a possible wrapper function for the system call and how the interaction with the tracer part could work. At the beginning of the application, an attempt is made to run the system call `open` for a file called

"framework.txt". As shown in Fig. 2, the tracer gets a notification that the application is about to perform a system call. `PTrace` now stores the information about the instruction pointer of the system call. The instruction pointer describes the position of execution within the tracee. In the next step, the instruction pointer is overwritten to the wrapper function. As a result, the tracee will execute the wrapper function defined in line 5 instead of the system call. Within the wrapper function, the application has the possibility to check the system call arguments by itself or by marshalling all arguments and contacting a broker service. Not only can the execution of the initially invoked system call be performed within the tracee, but also a broker service can be used to emulate it. When the tracee is designed to execute the same system call by itself, preventive measurements have to be considered to avoid that the application from entering an endless loop in attempting to perform the system call and getting rerouted to the wrapper function. This can generally be done using different ways:

1. The wrapper could be executed step-by-step until the return statement of the wrapper. As a result, the tracer would not get another notification in the default execution cycle of the tracer.

2. Keep track of the system call state within the tracer.

3. Set a break-point, similar to the working principle of debuggers, at the end of the system call to wait within the tracer for that specific signal to appear.

In all of the cases, the final problem is the same, which is to make sure that the tracee will continue its execution where it tried to perform the initial system call. The tracer can either manipulate the return jump address within the stack of the tracee or manually set the instruction pointer back to where it was. Using the latter approach, the tracer must also make certain that the stack frame is properly cleaned. Finally, the tracer can instruct the tracee to skip the system call and resume its execution. As a result, line 3 will return the resulting instruction pointer in read-only mode instead of read-write mode. This approach also comes with a couple of benefits and drawbacks described subsequently.

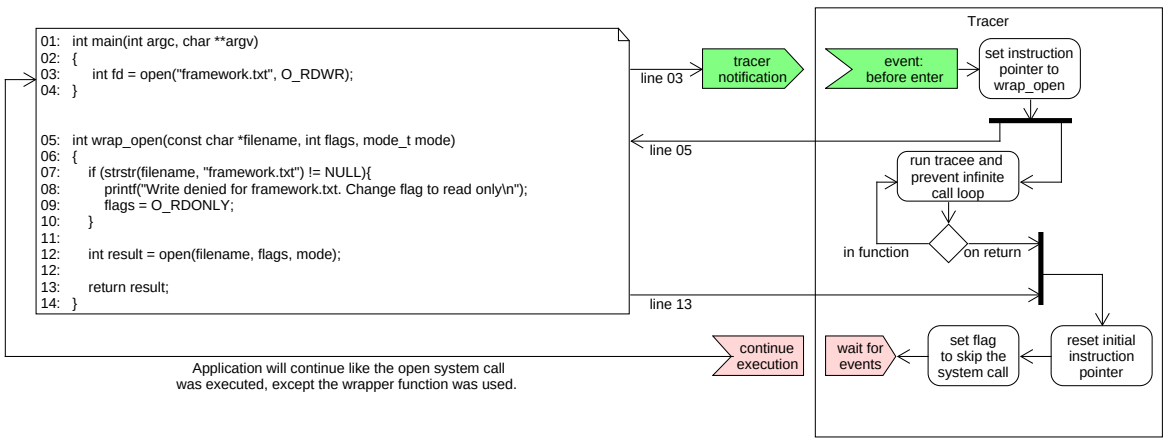

Fig. 3: Wrapper logic using `PTrace` for system call manipulations.

Privilege Separation and Concurrency

Depending on the concept, a pseudo privilege separation and dropping is built. For example, if the architecture is to check all arguments and return values in the tracee itself, an attacker with sufficient rights could either modify instructions or try to change data after they have been checked. The modification of instructions would be a rather sophisticated attack, but it is more

likely if the attack is launched within the same application, due to insufficient data validation, compared to as it would when launched from another one. Moreover, if the tracer does not ensure that all threads other than the one trying to perform the system call are stopped, multithreaded attacks can be launched. An attacker would, therefore, try to launch multiple threads. Within one thread, the intruder tries to open a file that he is allowed to use by using a shared buffer for the file name. In another thread, the file name within that buffer is constantly changed to a file that he should not be able to access. Due to the nature of threads to be interrupted at any time, there is a chance that the first thread trying to open the file is interrupted in line 11. If the second thread now changes the file name to the forbidden one, the intruder will be able to successfully open it. Using a broker service may partially solve this problem, depending on where the security measurements take place.

2.5.2 Tracer as Request Broker

Using the tracer as a kind of request broker to perform security checks on system calls is another way to deal with privilege dropping and separation. In this case, the idea starts as described in Sec. 2.5.1. When the main application, the tracee, tries to run the `open` system call, a notification is sent to the tracer. This time, the system call is not redirected to a wrapper function within the tracee, the tracer will do this job instead. Depending on the performed system call, the suitable wrapper method will be called. Now the tricky part using this approach is the simplified function calls in lines 3-5 and 10 of the tracer in Fig. 4. Contrary to the previously described method, there is no simple access to the system call arguments. It is necessary to differentiate between primitive data types and pointer structures. Simple data types can be read from the registers of the tracee using a suitable `PTrace` command. Nevertheless, it is essential to know and take into account that the registers where the values are stored depend on the architecture of the CPU and Linux. On 32 bit systems the first parameter is stored within the register `ebx`. On 64-bit systems, on the other hands, `rdi` is used. Regardless of these architectural details, pointer-based arguments are more challenging to retrieve and manipulate. For the read operation, for instance, the data address must first be retrieved from the register. In the second step, the data has to be read in chunks corresponding to the size of the system architecture. After all parameters are retrieved from the tracee, security checks can be performed. It is also possible to manipulate arguments before the system call is executed. The example in Fig. 4 overwrites the second parameter called flags in line 10. In the end, the tracer waits for new events and the tracee continues with the execution of the modified system call.

Benefits and Drawbacks

Compared to the redirection method in Sec. 2.5.1, multithreading is not a problem in terms of concurrent data manipulation as long as `PTrace` is configured correctly. If this is the case, `PTrace` halts all threads and child processes while it is running the event handler. This improvement, though, has a major drawback because the performance of the application will be drastically degraded. Security-wise, however, it is a large improvement since a possible vulnerability within the tracee makes it much more difficult to perform changes on the tracer due to the different memory space in which they are running. Implementation-wise, both approaches are challenging because they actively manipulate the tracee by changing either its instruction pointer or the arguments of a system call with its memory area modified. If erros occur, the tracee or tracer can be terminated unexpectedly, or data is overwritten if memory modifications are performed improperly. In any case, a `PTrace`-based solution should terminate the child application (tracee) if the tracer fails at any point. Otherwise, the child process may run, independently of its supervising process, and all system calls would be executed without any restrictions.

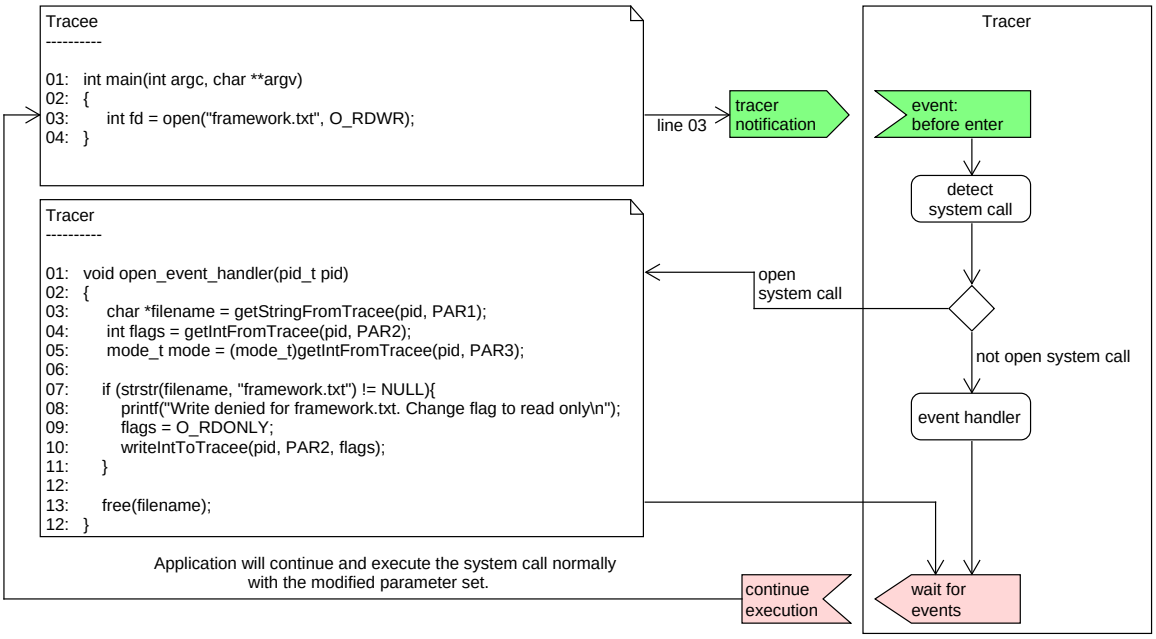

Fig. 4: Using a tracer as a request broker logic.

2.6 Architecture Evaluation

Various methods of privilege dropping and separation were described in the previous sections of this chapter. In combination with the primary task of this thesis in providing a simple toolchain to build filters based on `Seccomp` BPF and combining it within a request broker design so as to achieve privilege dropping and separation, the following rough architectures can be evaluated:

2.6.1 Model 1 / `LD_PRELOAD` Broker

The first model in Fig. 5 uses `LD_PRELOAD` described in Sec. 2.3 to overwrite the original system calls to implement the aspect of privilege dropping. The requirement of privilege separation is achieved by launching a request broker with the main application. The wrapper functions will primarily marshal the arguments and pass them to the request broker, which decides if a system call is allowed. Depending on the system call, the execution can then be performed either on the broker or on the main application side.

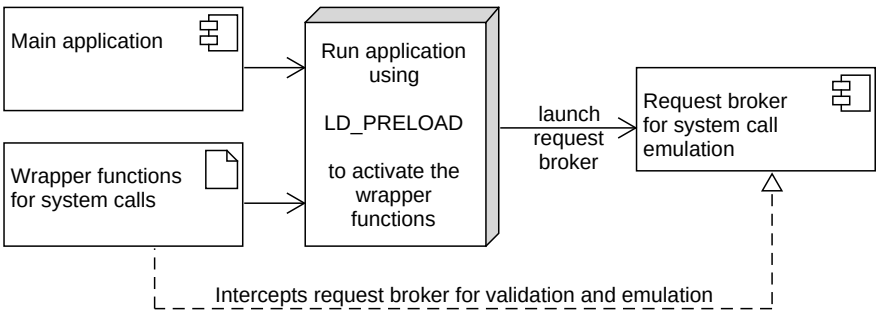

Fig. 5: Model 1 using `LD_PRELOAD` with a request broker.

2.6.2 Model 2 / `Seccomp` Trap Broker

The second model in Fig. 6 uses `Seccomp` BPF-policies to directly check system calls on the kernel level. The involved actions, such as system call manipulations and the inspection of pointer-based arguments are performed within the trap functions (signal handlers) called by `Seccomp`. These trap functions may then call a request broker to perform necessary actions and validation checks.

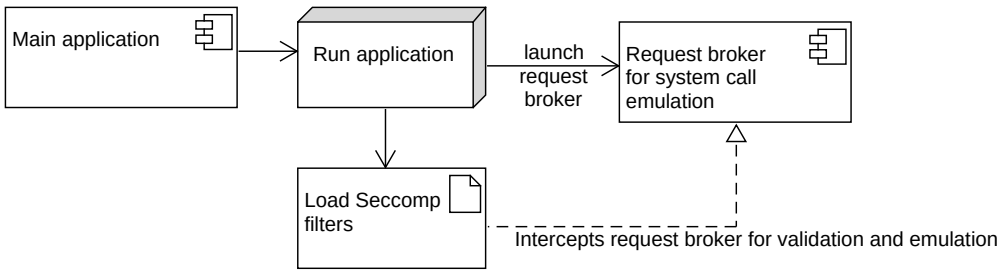

Fig. 6: Model 2 using `Seccomp` traps with a request broker.

2.6.3 Model 3 / `PTrace`

The third model in Fig. 7 uses `PTrace` described in Sec. 2.5. It is not specified weather the request broker is an external process or integrated with the tracer part of the application. It not defined further if system calls are redirected to a wrapper within the target application or the tracer itself. On the first run of the application, a fork takes place. The parent process becomes the tracer and the child process the tracee, which directly executes the same application again. On the second run, a check must be performed if the process is already observed by a tracer instance. If this is the case, the execution of the main source is started.

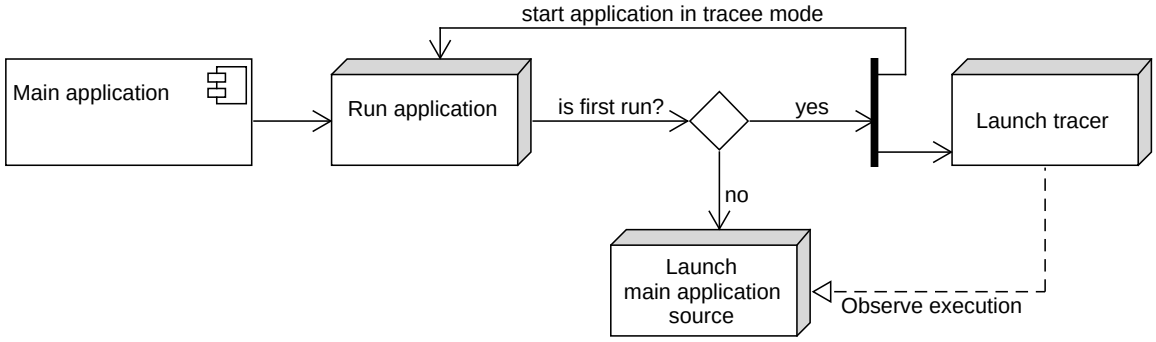

Fig. 7: Model 3 using `PTrace` with build in broker.

2.6.4 Model 4 / `PTrace` Combined with `Seccomp`

The fourth and last model in Fig. 8 combines the `PTrace` module described in Sec. 2.5 and `Seccomp` described in Sec. 2.4. The start process is exactly same as described in Model 3. The only change is that both the tracer and the tracee will load a set of `Seccomp` rules to get rid of excessive permissions. Contrary to Model 2, the filter policies of the tracee in this model will send a signal to the tracer instead of a trap function.

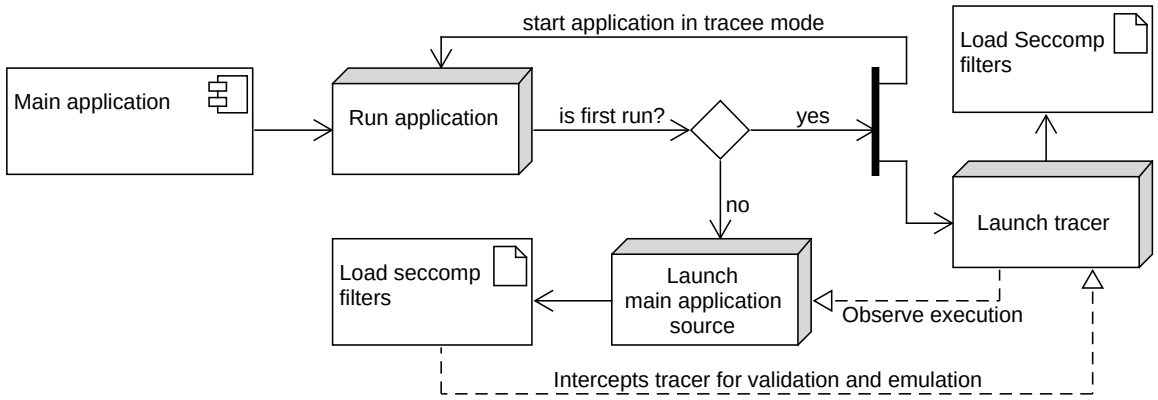

Fig. 8: Model 4 using `PTrace` combined with `Seccomp`.

2.6.5 Comparison

To evaluate the most suitable method for the target application of this thesis, all four methods described in the section Architecture lineup, Sec. 2.6, are compared with each other within multiple categories:

Complete Interception

The most important aspect is the ability to intercept all system calls regardless of their originating call location. Because Model 1 is based on `LD_PRELOAD`, it will not be considered for further comparisons. As described in Sec. 2.3, this method can only intercept function calls that are defined in a linked library. When a library is statically linked or the application directly communicates with the kernel to run a system call, `LD_PRELOAD` will not be able to intercept it. The other methods from 2-4 are all capable of intercepting system calls at the core level.

Wrapper Limitations

To define complex sanity checks for system calls, it is important that there are no limitations to the functions that can be called within a wrapper function. Conclusively, Model 2 is also not a suitable solution for the target framework of this thesis. This is because the trap function called by the second model is executed within a signal handler, which has the limitation that only async-signal-safe functions should be called. This drastically reduces the possibilities within a wrapper function because space cannot be allocated dynamically because `malloc` and `mmap` do not have this characteristic. The other two methods have no limitations regarding this aspect.

Low-Level Checks

The solution should be able to permanently drop privileges at the core level. This means that even if the main application is compromised, there is no possibility to regain access to the dropped privileges. This is where `Seccomp` comes in. Once the `Seccomp`-based Berkeley packet filters are loaded into the kernel, there is no way to modify them as long as the tracer part of the application is properly configured. Conclusively, Model 4 using `PTrace` with `Seccomp` describes the rough structure of the final framework.

Performance and Multithreading

In all solutions, the performance in multithreaded applications is reduced. The overhead is caused by many factors: marshaling and unmarshaling system call arguments if an external broker service is used. Halting all threads while `PTrace` handles an event. Executing `Seccomp`-based Berkeley packet filters in front of every system call. In fact, any additional check of a system

call and its arguments before and after its execution causes an overhead. As a consequence, these drawbacks will be ignored in the first implementation of the framework. The main focus lies on the security aspect.

Security

The final framework should have a high-security level. Therefore, all security checks should be performed within the kernel using `Seccomp` where this is possible. To minimize the complexity and error-proneness of the framework, the tracer will be used as a request broker. Using a third process would not introduce a significant boost in security because the tracer would still be the single point of failure and the overall performance would be further degraded. Nevertheless, it is important that the tracer part of the application also has limited access to system resources and system calls. Thus, `Seccomp` should also be used for this part of the application.

Final Verdict

Based on the framework requirements and the architecture comparisons in Sec. 2.6.5, Model 4 will be used. It allows `Seccomp` to be used to perform system call checks within the kernel and integration with `PTrace`. The tracer will constantly observe the tracee and perform sanity checks based on a defined rule set. Model 1 must not be used because only functions can be intercepted instead of system calls. Model 2 has severe limitations to function calls within the main trap function to perform sanity checks. Model 3 does not use `Seccomp` to validate a system call, and its arguments are based on primitive data types within the kernel.

3 Framework

The framework built in this work consists of two main modules:

1. A Python-script to generate `Seccomp` and C-based wrapper rules out of a configuration file.
2. The main framework for C-based applications.

The interaction between these two components can be seen in Fig. 9. The first part is represented by the configuration builder transforming three configuration files to C code used by the main framework. Each configuration file serves a particular purpose:

- Rule configuration:
  defines which system calls are allowed under consideration of their argument data. This file also offers attributes to configure the overall framework.
- System call configuration:
  C-based source file defining the prototypes of the system calls. It allows functionalities to be implemented beyond the capabilities of the rule configuration file.
- Templates:
  source templates used by the configuration builder to construct the final C files. This configuration file must not be modified by any user.

The first set of output files consists of the C code and header file for the functions that are responsible to load the `Seccomp` rules for the tracer and tracee into the kernel. The second C code and header file is the wrapper function source. It defines the validation checks and system call manipulations that must be performed within the tracer part of the application. Once the files are generated and linked to the main application together with the rest of the C-based framework, a single function call is sufficient to secure the application.

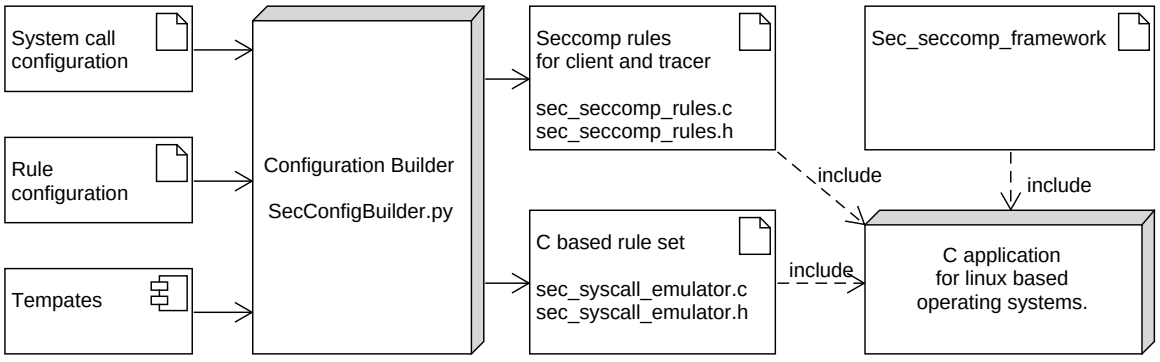

Fig. 9: Framework overview.

The following sections provide a deeper insight into the overall architecture and modules.

3.1 Architecture overview

The architecture is based on Model 4 described Sec. 2.6. If `Seccomp` is used in combination with `PTrace`, the tracer can normally intercept a system call at three different states as shown in Fig. 2. In this case, the first invocation is redundant to the second one. As a result, only the trigger event by `Seccomp` and the second trigger after the execution of a system call are used. Fig. (10) demonstrates the execution path of a system call in the final implementation.

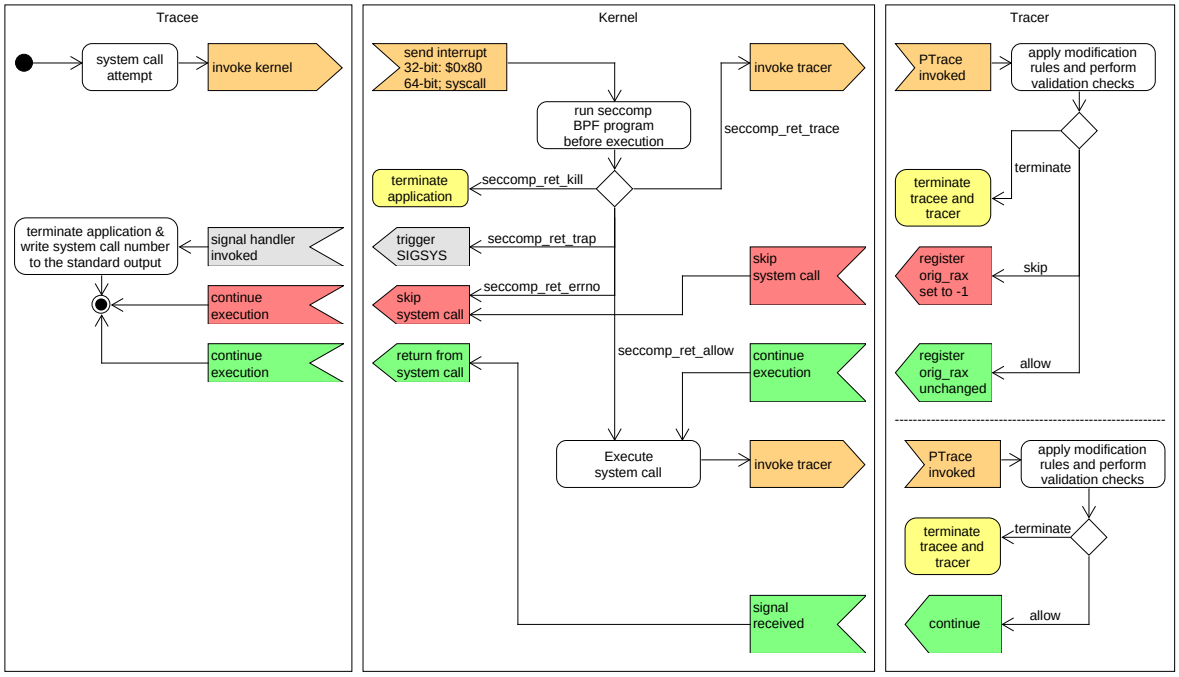

Fig. 10: Framework system call execution.

The process begins when the tracee tries to perform a system call. This invokes the kernel which happens in a different way according to the system architecture. On 32-bit based systems, an interrupt with the value *"0x80"* is triggered. On 64-bit systems, on the other hand, an assembly instruction `syscall` is used to invoke the kernel. In both cases, the `Seccomp` filter program will be executed. Depending on the outcome of the BPF, four execution paths are differentiated. If *`SECCOMP_RET_KILL`* is triggered, the application gets terminated immediately. *`SECCOMP_RET_TRAP`* invokes a *`SIGSYS`* signal that calls a trap handler. In the current implementation, the trap handler is used for debugging purposes. It allows a system call number to be printed, showing what system call caused the application to fail. If the return value is *`SECCOMP_RET_ERRNO`*, the system call is not executed and the C-based variable `errno` is set to *`ENOSYS`*, indicating that the system call failed due to the missing implementation. This error code is also used by `PTrace` if a system call is skipped. *`SECCOMP_RET_ALLOW`* instructs the kernel to execute the system call normally. The last execution path is represented by *`SECCOMP_RET_TRACE`* that invokes the tracer of the application.

With `PTrace`, the tracer waits for this event. After its invocation, the multiplexer of the tracer calls the correct wrapper function based on the system call number. The wrapper function first reads the argument data from the registers and memory space of the tracee. Afterward, the system call arguments can be compared to the specific values with the ability to modify them. Regardless of the complexity of the wrapper function, three different outcomes can occur. *`Terminate`* instructs the tracer to immediately stop the execution of the tracee. *`Skip`* overwrites the register `orig_rax` to *"-1"*, defining that no system call should be executed. Depending on the

return value and argument data, the system call will be interpreted in the main application as a success (emulation) or failure. *Allow*, on the other hands, keeps `orig_rax` unchanged. In case of *Allow* and *Skip* the tracer returns from its invocation and the execution path continues within the kernel. If `orig_rax` is modified to *"-1"*, the system call wont be executed, which is recognized as failed execution within the main application. When the `orig_rax` register is unchanged, the system call will be executed.

After the execution of the system call, another invocation of the tracer is made. This time, the same procedure happens again. The difference now is that the data of the executed system call can be checked and modified. For instance, this allows the data received by the system call `recvmsg` to be read and manipulated before it is returned to the callee of the system call. It would, therefore, be possible to change the data that an application reads from a file. Contrary to the first invocation, only two return states are supported at this stage. First, there is *terminate*, which immediately stops the execution of the application. *Allow* returns the focus to the kernel, which then shifts it further to the main application. After that, the tracer continues code execution until the next attempt to execute a system call is made.

Debug Mode

The framework supports a debug mode, which is sometimes needed to understand which system call leads to the termination of an application. If the debug flag is set in the configuration file, the `Seccomp` actions will be emulated within the tracer. Hence, the BPF will call *SECCOMP_RET_TRACE* with custom flags for the action instead of *allow*, *skip* and *kill*. The tracer differentiates between these custom flags, prints debug information and emulates the action that `Seccomp` would have performed.

3.1.1 Application of `Seccomp`

`Seccomp` is one of the core modules used within the implemented framework. The target function of `Seccomp` is the ability to perform system call checks within the kernel and permanently drop privileges. With C, there are two common used ways to create `Seccomp` filters:

1. Directly write BPF-instructions as they are shown in Algo. 2.
2. Use an abstraction library like libseccomp.

libseccomp which is more frequently used allows rules to be defined in a simple, readable manner. The library then automatically translates the abstract rule format into Berkley packet filters. For instance, Function `loadClientSeccompRules`, as is shown in Algo. 4, demonstrates what the same filter set as shown in Algo. 2 would look like using the abstraction library.

libseccomp was used in the initial implementation. It had to be replaced by the custom implementation because libseccomp sometimes optimized rules in such a way that its behavior changed. This behavior can be demonstrated using a simple rule set as is shown in Algo. 5. The filter basically checks different argument combinations of the system call `socket`. According to line 3-5 the system call is allowed when the domain is set to `AF_UNIX` with a type definition of `SOCK_STREAM`. If the domain type is set to `AF_INET` instead, the application must be terminated. In case neither of the previous rules apply, related to the system call `socket`, the tracer must be informed of performing further checks to make the final decision. All other system calls will lead to a termination of the application according to the default rule in line 2. However, libseccomp produces the output as shown in Algo. 6. The first two rules are ignored because there is a general rule for the system call `socket`. This is not the expected behavior. The objective is to source out all checks on `Seccomp` where this is possible. Security should be the most important aspect; hence, it is `Seccomp`, not the tracer that ought to be the primary wall of defense.

```
void loadClientSeccompRules(){
  if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("Could not start seccomp:");
    exit(1);
  }

  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_KILL);
  // Add general allow rules
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);
  // Add specific allow rules
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
              SCMP_A0(SCMP_CMP_EQ, AF_LOCAL),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM))
    ;
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
              SCMP_A0(SCMP_CMP_EQ, AF_UNIX),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
  // Add specific skip rules
  seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EPERM), SCMP_SYS(setrlimit), 1,
              SCMP_A0(SCMP_CMP_EQ, RLIMIT_CPU));
  // Add general modify rules
  seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(open), 0);

  if (seccomp_load(ctx)== -1) {
    perror("\method{Seccomp} could not be initialized. Abort Process.");
    exit(1);
  }
}
```

Algo. 4: libseccomp filter example.

```
scmp_filter_ctx ctx;
ctx = seccomp_init(SCMP_ACT_KILL);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
                 SCMP_A0(SCMP_CMP_EQ, AF_LOCAL),
                 SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(socket), 2,
                 SCMP_A0(SCMP_CMP_EQ, AF_INET),
                 SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(socket), 0);
```

Algo. 5: libseccomp filter before optimization.

As mentioned, a new abstraction library had to be implemented, translating the rules into BPF without changing the expected behavior. Because libseccomp is widely used and the user should easily understand the generated rules, the implemented solution provides the interface demonstrated in Algo. 7. The newly implemented `Seccomp` BPF generator also optimizes some parts of the rule set to optimize for optimal performance, but it does so without changing the expected behavior. Unlike its role model, the new implementation does not have the full scope of operation and fewer system architectures are currently supported.

Tracer and Tracee

`Seccomp` rules are generated for both the tracer and tracee. If the tracer were somehow be compromised, this would help to reduce the impact on the system, even though there is no direct communication between the tracer and tracee. In general, the tracer has the exact same filter rules as the tracee. Giving the tracer less permission would yield no benefits in terms of

```
#
# pseudo filter code start
#
# filter for arch x86_64 (3221225534)
if ($arch == 3221225534)
  # filter for syscall "socket" (41) [priority: 65535]
  if ($syscall == 41)
    action TRACE(16);
  # default action
  action KILL;
# invalid architecture action
action KILL;
#
# pseudo filter code end
#
```

Algo. 6: libseccomp filter after optimization.

```
seccomp_ctx ctx;
ctx = sec_seccomp_init(SCMP_ACT_KILL);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
                     SCMP_A0(SCMP_CMP_EQ, AF_LOCAL),
                     SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
sec_seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(socket), 2,
                     SCMP_A0(SCMP_CMP_EQ, AF_INET),
                     SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(socket), 0);
```

Algo. 7: `Seccomp` BPF generation framework.

security since an attacker could just execute the malicious code within the tracee. Furthermore, the tracer may need this permission if the tracee defines a `Seccomp` filter, notifying the tracer to emulate the system call. It is usually the case that the tracer will have slightly more rights, as it is the tracer that must have access to system calls that control and manipulate the child process. Within the framework, the tracer's filter rules are copied by the tracee and slightly modified by extending them with a custom rule set.

Threads and Forks

During the application life cycle, it is likely that either new threads will be created and the process will be forked or applications will be launched using `execve`. These are all critical moments concerning the security of an application and the overall system. If configured properly, `Seccomp` automatically mitigates most of the possible security loopholes which could be introduced by these actions. For all these events, the filters will automatically be copied in the new thread or process. It is, therefore, impossible for an attacker to simply launch another application and run privileged commands on behalf of that. Another special characteristic is that the `Seccomp` return value *`SECCOMP_RET_TRACE`* that notifies the tracer would fail if no tracer is present. To activate these security measurements, the `Seccomp` flags `SECCOMP_FILTER_FLAG_TSYNC` and `SECCOMP_SET_MODE_FILTER` have to be set when the filters are loaded into the kernel.

Logging

With `Seccomp` rules defined, it is normally difficult to grasp which system call leads to the termination of an application, especially if the framework is integrated into an existing application. If the debug flag is set in the rule configuration file, two logging mechanisms are activated. The first mechanism depends on the used kernel version of Linux. If a version newer than 4.14 is used, the flag *`SECCOMP_FILTER_FLAG_LOG`* will be activated. It automatically logs `Seccomp`-based actions into the file *"/proc/sys/kernel/seccomp/actions_logged"*. The second mechanism

reroutes all `Seccomp` actions like ***kill***, ***allow*** and ***skip*** to the tracer, which would normally be executed within the kernel. The tracer then logs the action, using the Linux-integrated syslog library. This library automatically writes information about the `Seccomp` filter action to the file *"/var/log/syslog"*. The defined rules in Algo. 4 are transformed into the ones as shown in Algo. 8. The third option that can be used to make debugging easier without changing the debug flag is simply to set the default action to ***SCMP_ACT_TRAP***. The framework's trap function, which prints information about the system call for the standard output, is called before the application is terminated.

```
void loadClientSeccompRules(){
  if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("Could not start seccomp:");
    exit(1);
  }

  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_TRACE(PTRACE_DBG_TERMINATE));
  // Add general allow rules
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(
    exit_group), 0);
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(exit),
    0);
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(read),
    0);
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(write),
    0);
  // Add specific allow rules
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(socket),
     2, SCMP_A0(SCMP_CMP_EQ, AF_LOCAL),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_ALLOW), SCMP_SYS(socket),
     2, SCMP_A0(SCMP_CMP_EQ, AF_UNIX),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
  // Add specific skip rules
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE((EPERM << PTRACE_DATA_SHIFT) |
    PTRACE_DBG_SKIP), SCMP_SYS(setrlimit), 1, SCMP_A0(SCMP_CMP_EQ, RLIMIT_CPU))
    ;
  // Add general modify rules
  sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_DBG_MODIFY), SCMP_SYS(open),
    0);


  if (seccomp_load(ctx)== -1) {
    perror("\method{Seccomp} could not be initialized. Abort Process.");
    exit(1);
  }
}
```

Algo. 8: `Seccomp` filters in debug mode.

Abstraction Layer

To be able to generate the barely readable Berkeley packet filter instruction, an abstraction layer was implemented as mentioned in Sec. 3.1.1. The general idea is that the user can quickly compare the initially defined rules within in the rule configuration file against the generated `Seccomp` rules. This would be hardly practicable if the BPF rules were generated directly by the Python script. Another benefit is the ability to simply extend the abstraction layer with new functionalities. `Seccomp` Berkeley packet filters do not support the comparison operators `not equal`, `lower than` and `lower or equal than`. These have to be modified into their counterparts `equal`, `greater or equal than` and `greater than`. When it comes to implementation, the abstraction layer is kept simple to minimalize the source of errors. Besides, the generated BPF rules can be exported to see what is loaded into the kernel.

3.1.2 Application of `PTrace`

`PTrace` is used to extend the functionality of `Seccomp` and overcome its limitations. This implies it has the capability to perform checks on pointer-based system call arguments, the parameter manipulation, and the return value. `PTrace` further allows the checks on the argments and the return value to be performed even after a system call is executed. Several characteristics of `PTrace` and Linux itself need be to considered. The most important are described in the subsequent paragraphs.

System Call State

Fig. 2 visualizes `PTrace` in combination with `Seccomp`, which can detect a system call under three states. At least two of the states must be differentiated differentiated, one being the `Seccomp`-based trace event and the other being the event after the system call is executed. For this, the tracer stores a list of all system call states for each calling process and thread. This is especially important because the user not only can define the rules that should be applied either only before or after the system call execution but also can carry out a combined check. To perform the correct action successfully, the state information is crucial. Just updating the state primitively on each call is not sufficient and can result in a wrong state detection because kernel versions 3.5 to 4.7 behave differently from the newer versions or those ones equal to version 4.8. In the previous versions, the `Seccomp` event occurred before the entry and exit stops of a system call. After version 4.8, this behavior changed so that the `Seccomp` event was triggered between the entry and exit stop. This is a vital issue because the tracer is unable to distinguish between an entry- and exit-stop. Besides, there are cases where additional events are generated. If `execve` is executed, four events are generated instead of three. What comes out first is the entry-stop and the rest are the `Seccomp` event, the exit-stop, and a notification that an `exec` call has occurred. Due to these circumstances, the framework is limited to the kernel versions that are greater or equal to version 4.8. This makes it possible to distinguish precisely between the `Seccomp`-triggered event and the exit stop that comes right after.

System Call Invalidation

`Seccomp` offers a rule action, that permits the execution of a system call to be skipped and the specific error number to be returned. Because `PTrace` is seen as an extension of `Seccomp` in this framework, the tracer must also support this capability. This functionality can only be emulated to a certain degree.

Every application written in C can import a library called `errno.h`. This library defines a global variable called `errno`, which is used by system call wrapper functions and library functions, to return the error codes describing what went wrong if they fail. Because these error numbers are part of the C language and not the kernel itself, system calls cannot set this variable directly. If an error occurs, they return the error number in the same register as they would return a valid return value. However, the system call `fork` creating a copy of the running process is an exception. This system call writes error codes into a separate register. Regardless of these special cases, `errno` is kept untouched. Here the wrapper functions from the standard C-libraries come into action. With respect to `write`, the wrapper function validates both, the arguments and the return value, and then sets the `errno` variable based on these data.

The return value *`SECCOMP_RET_ERRNO`* of `Seccomp` writes the defined error code directly into the *`errno`* variable even if the kernel is directly interrupted in order to execute a system call. Two methods exist for the tracer to emulate this behavior:

1. The tracee could send his address of `errno` to the tracer. The tracer could then modify its value by manipulating the memory space of the child process. This comes with a significant performance overhead.

2. Overwrite the register `orig_rax`, on `x86_64` system architectures, defining the executing system call number to *"-1"*.

The second option the used method in this framework. Changing the executing system call number to *"-1"* instructs the kernel to skip the system call as `Seccomp` would do it. The only difference is that this will always set the errno variable to ***ENOSYS***. This error code defines that the function is not implemented. Thus, it is not possible to adjust this value to a more suitable error code like ***EPERM***, defining that the call failed due to insufficient permission. Therefore, the framework is configured in such a way that the `Seccomp` action ***SECCOMP_RET_ERRNO*** returns the same error code ***ENOSYS***.

Registers

Registers are small and quickly accessible storage elements used by a CPU. All arithmetic operations are directly executed on data within these registers. If an application intends to perform a multiplication of two numbers, these numbers are first loaded into different registers. Subsequently, the CPU calculates the result and stores it in another register. If data should be permanently available, it is written into the long-term memory area of a computer. This background information is required to understand how system call arguments can be retrieved from the tracer and what kind of challenges occur when they are manipulated. The name of the registers depends on the architecture as listed in Tab. 1. `x86` represents 32-Bit architectures and `x86_64` represents 64-Bit architectures with support for 32-Bit applications. Due to backwards compatibility, 64-Bit systems also support the old registers from the `x86` architecture, but not the other way round.

			Parameters					
Architecture	**Return value**	**System call number**	**1**	**2**	**3**	**4**	**5**	**6**
x86	eax	eax	ebx	ecx	edx	esi	edi	ebp
x86_64	rax	rax	rdi	rsi	rdx	r10	r8	r9

Tab. 1: System call registers.

An `x86_64` application invoking the `open` system call would store the number defining the system call to be executed in the register `rax`. The file name, which is the first parameter, is stored in the register `rdi`. The access flags belong to the register `rsi` and the access mode can be found in the register `rdx`. `Seccomp`-based filters will check the values in these registers exactly. It is important to note that the first parameter of `open` is a string specifying the name of the file to `open`. Because these registers are limited in size, it is not the string that is stored within the register `rdi`. It will contain only a pointer to a memory region in the application defining the string. This is also the reason why `Seccomp` is not able to perform detailed checks on such data.

In order to access the data of the system call from the tracer, these register values must be retrieved from the tracee. Primitive data types such as integers and floating-point numbers are therefore not problematic because `PTrace` offers a single instruction to acquire those data.

Pointer-Based Arguments

To retrieve the data of a child process behind a pointer, the pointer address must be retrieved from the register and the data must be read byte-wise from the tracee under the consideration of the system's byte-endianness. The data cannot always be read the same way. A system call such as `write` has three parameters: the file descriptor, the pointer to a buffer containing the data, and the variable defining the size of the data to be written. In such a case, the framework must first be able to extract the size from the last parameter before the buffer can be read. Otherwise,

the framework would not know the length of the data the application intends to write. The situation looks different for a system call such as `open`. There is no parameter to define the size of the argument describing the path and name of the file to which an open attempt is made. In such a case, the tracer, as well as the system call, rely on the correctness of the arguments. Both will start reading from the passed address until the special character *"\0"* is found, defining the end of a string.

Much more complex than just reading the data behind a pointer argument is their manipulation. In modification scenarios, the script's differentiation between the two types of pointer-based arguments is crucial.

The first argument type is called return argument. Return arguments are used in system calls such as `read`, `gettimeofday` and `getcwd`. These system calls share the property that the calling application reserves memory space, whose location then is passed to the system call. The system call then prepares the data and writes it to the defined location. As with any system call, the tracer must ensure the integrity of the memory area by not writing more data than what the application initially reserved. Further, it must be guaranteed that overlapping memory regions are kept consistent, because, on 32-Bit systems, `PTrace` is only able to overwrite data with the same length as the architecture itself. To better visualize the possible problems, Tab. 2 illustrates what a memory region can look like. Suppose that two strings are stored consecutively. The first one contains *"ABCDEFGHIJ"*, the second *"12345678"*. The character *"\0"* represents the end of each string. Each letter has the size of a byte (4 bits). DWORD labels the regions that can be read with `PTrace`. This means that one can access any DWORD 1-5 consisting of 32-Bits each. If the first string should now be changed to *"BCDEFGHIJK"*, the tracer must read all double word blocks, especially the number 3, modify them byte-wise, and write the new block back to the memory region. Without first reading the data blocks, the string 2 could also be modified unintentionally. This is because the first number is stored in the same block as the last letters of the first string.

String 1 = *"ABCDEFGHIJ"*											**String 2 = *"1234"***							
DWORD 1				DWORD 2				DWORD 3			DWORD 4				DWORD 5			
A	B	C	D	E	F	G	H	I	J	\0	1	2	3	4	5	6	7	8	\0

Tab. 2: Data replacement example.

The second pointer-based argument type is used by system calls such as `open`, `chdir` and `setrlimit`. These functions share the characteristic that the calling application also reserves memory for the data structures used. However, unlike the previously described argument type, the data structure will not be modified by the system call. The function only uses the data for execution. Modifying these arguments must be done differently for the following reasons. The data may be within a read-only region of the memory or it may be used by other threads. Therefore, modifying the same memory region is not possible if one wants to replace the specific string with a larger one. A possible solution would be to reserve and use memory in the client application. As simple as it sounds, the tracer would first have to evaluate where the new memory region should be created, and after the execution of the system call, the modified memory region must be cleaned up. The second possibility is to use a shared memory region between the tracer and tracee. This approach has the drawback that the shared memory region could be accessed according to the user or group ID with which the application is running. This means that other applications may be able to extract confidential data if the application does not run with a separate user or group ID. The third option involves using the stack. Note that system calls are not executed with the stack of the application. The kernel stack is used instead. It temporarily stores data for function calls. Consequently, read and write operations are permitted. According

to the AMD64 ABI for `x86_64` systems, modifications of the stack are permitted under certain restrictions [18]. Specifically, 128 bytes, the so-called red-zone, must be kept intact. This zone is located directly below the stack pointer. The further advantage is that this space must not be cleared manually. Hence, all the modified pointer arguments are stored there. To pass this new data area to the system call, the pointer in the argument-related register must be overwritten. Besides the advantages over the other methods, the risk of using too much space of the stack is minor due to the stack sizes used today.

Multithreading

Multithreading is widely used in today's applications. It enhances the overall performance of an application because tasks can be executed in parallel. The performance, however, is reduced if system calls are constantly made and a tracer is attached to the threads. This is related to the existence of only one observer process. When a thread or child process tries to execute a system call, the tracer gets notified and all traced objects will be halted. Otherwise, the tracer would not be able to intercept all system calls. Unlike to the performance issues, the security aspect of it benefits from this behavior. For instance, if two threads share a memory space, one thread can store a valid file name, that it is allowed to open. The second thread can change the value permanently to a forbidden file. If the tracer in this scenario checks the old valid name, permission is granted. If the second thread is able to modify the data right after the validation check took place, a non-permitted file can be accessed. However, there is a way to prevent this behavior. The tracer can always copy the data in the stack and pass those arguments, which might result in further performance issues and a stack overflow.

3.2 Limitations

The implemented framework has several restrictions due to the chosen architecture, Linux itself, and the current progress of its realization.

File Descriptors

Many system calls use file descriptors to access different types of resources and devices in a Linux-based operating system. The first limitation concerning this, however, is the inability to modify them within the tracer. Therefore, the tracer cannot emulate the `open` system call. All that can be done is to check and manipulate the file name before permission is granted or denied. The reason for this is that file descriptors belong to a specific process and are tightly bound to its permissions. Even though various processes can pass file descriptors between one another, `PTrace` does not directly support its injection into a child process. The only way in which they can be exchanged is through sockets. This means that the tracer would have to send the file descriptor and the tracee must receive it actively. There is no way of attaching it to a tracee behind the back without modifying the kernel by the specific module, which would harm the portability of the framework. To circumvent this issue, the tracer could manipulate the execution flow of the tracee so that a prepared function to receive the file descriptor is executed before the application continues one instruction ahead of the initial `open` call. This particular solution is not usually considered because there is no real security benefit. Hence, the tracer is the single point of failure in both described cases. This approach would unnecessarily increase the complexity of the solution.

A further limitation is the ability to check the path behind a file descriptor. This affects, for example, the rules in which the system call `write` should be limited to the specific set of files. In such a case, it is not guaranteed that the rule works as expected owing to the implementation of file descriptors themselves. File descriptors have the characteristic of either having zero, one or multiple file names attached. No file name is available if the object represents a stream or the standard output of an application. Multiple file names are available when dealing with hard links. This means that a file can be accessed by multiply different paths. A rule that

blocks access to files within the directory *"~/desktop/blocked"* and a rule that grants access to files within *"~/desktop/allowed"* can be used to illustrate the problem of hard links. It is then possible to create a hard link *"~/desktop/allowed/forbidden.txt"* which points to the file *"~/desktop/forbidden/passwords.txt"*. Prioritization implies that one of the rules is applied. More severe is the challenge to obtain the paths; one possibility is to scan the entire file system recursively and try to find the inode number of the file descriptor. This is not practicable. It is further possible that multiple files have the same inode number, but not vice versa. This then reduces to the problem of obtaining the proper file. For this matter, a simplified method has been adopted into the framework. It is possible to check the path of a file descriptor if the object is stored in *"/proc/(pid)/fd/(file descriptor number)"*. Nevertheless, the user must still keep in mind that the provided path can be obsolete at any time.

Performance

Any solution with a focus on security aspects will show degraded performance. To measure the extent of degradation caused by the framework, measurements are performed at the execution time of a range of system calls. The used Debian test platform consists of a virtual machine running on an Intel Core i7-4700HQ at 2GH with two cores assigned. Each system call has been executed five million times in a row on five different configurations:

- Without framework integration.
- Permit execution using a `Seccomp` rule.
- Disallow execution using a `Seccomp` rule.
- Invoke the tracer using a `Seccomp` rule and permit the execution of the system call.
- Invoke the tracer using a `Seccomp` rule and disallow the execution of the system call.

Fig. 11 shows the result of the performance measurements. In particular, the calls `getrlimit`, `getpid`, `getuid`, `getcpu`, `getegid`, `open` and `close` are analysed and described subsequently:

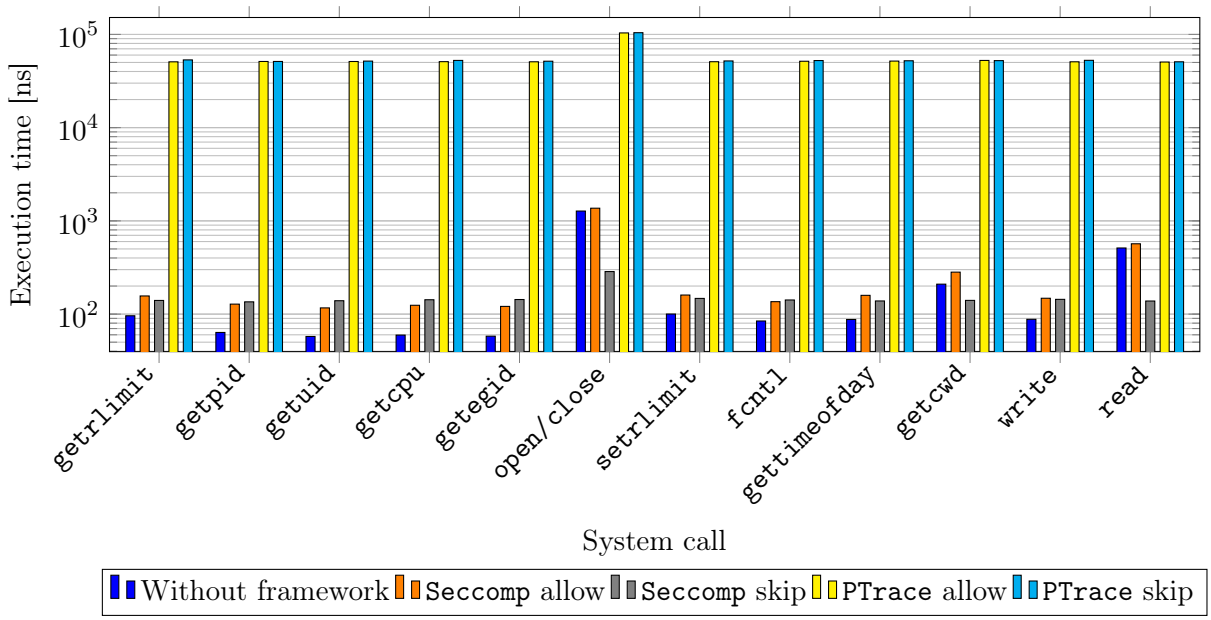

Fig. 11: Framework performance measurements.

- `getrlimit`:
  is often used to obtain information about certain resource restrictions such as the number of allowed processes, the available execution time, maximal stack size and others. The overhead for the `Seccomp`-based checks ranges from 46-63%. Decisions in the space of the tracer usually take about 108 times longer than an immediate execution.

- `getpid`:
  is a modest call that returns information about the ID of a process. `Seccomp` introduces an overhead of 101-112%. `PTrace` further increases the duration of the execution by approximately 75 times.

- `getuid`:
  returns the ID of the user running the application. The use of `Seccomp` leads to an overhead of about 102-141%. `PTrace` is 177 times slower compared to an unchecked invocation.

- `getcpu`:
  is often used by applications that aim for CPU-based optimizations. The execution time of this system call depends strongly on the architecture used. If `Seccomp`-based checks are performed, an execution time overhead of about 108-139% must be expected. When `PTrace` is used, the execution becomes an additional 73-81 times slower.

- `getegid`:
  returns the effective group ID of the applications and is implemented similarly to `getuid`. As a result, the impact on performance is the same.

- `open`/`close`:
  describes a scenario where a file was continuously opened and closed. Due to the higher complexity of these functions compared to the others, the normal execution time is nearly doubled. For these system calls `Seccomp` causes an overhead of only 7.2%, which is a good result because they are usually called multiple times within an application. Validation checks within the tracer result in an overall overhead of 16.2%.

- `setrlimit`:
  has the opposite functionality to `getrlimit`. Permission checks on the kernel level result in an overhead between 47-59%. `PTrace` leads to 101 times slower execution time.

- `fcntl`:
  can be used to get and set properties of a file descriptor. The `Seccomp` overhead ranges from 61-68%. `PTrace` increases the execution time by a factor of about 123.

- `gettimeofday`:
  is a frequently used system call to obtain information about the current date and time zone. The overhead for `Seccomp` is between 58-81%. `PTrace` further degrades performance by a factor between 65 and 75.

- `getcwd`:
  returns the current working directory of an application. The overhead of `Seccomp` is 34% if it is permitted, and -33% in case permission is denied because less time consuming actions are required. `PTrace` on the other hand has nearly a constant performance degradation by a factor of 50.

- `write`:
  is a commonly used system call. It is used every time when data is written towards a file descriptor. Each write call results in an overhead of about 65% when `Seccomp` is used and a factor of 117 when `PTrace` comes into action.

- `read`:
  is like write a commonly used system call in applications to obtain data from a file descriptor. Depending on the action, the overhead is about 11% or -73% with `Seccomp`. `PTrace` increases execution by a factor of 18.

Conclusion

The overhead of the framework depends heavily on the system call. Not all system calls lead to the same degradation of performance. The few examples listed in Fig. 11 show that more complex system calls tend to have a lower overhead compared to others. In average, `Seccomp` results in an overhead of about 60.3%. `PTrace`, on the other hand, increases the average execution time by a factor of 87. If security and not time are a crucial for an application, the framework developed in this thesis offers powerful functionalities to restrict access to the system and its resources. In addition, an application that rarely uses system calls will barely be touched by the impact of the reduced performance. Also in other cases, the performance may be much better than expected as also demonstrated in the application of the framework for the `Nginx` web server in Sec. 4. The overhead for processing requests checked by `PTrace` resulted in an overall performance degradation of 115.78%.

3.3 Configuration Language

The configuration of the framework consists of two files. One is a C-based source file defining the system call prototypes together with some annotations to specify the special behavior of some arguments. The other is *"ini"* file describing the rule set for the system calls.

3.3.1 System Call Prototypes

The C-based configuration file with the default name *"sec_ syscalls_ conf.c"* contains the following information:

- system call arguments with their type,
- dependencies among single arguments,
- group names for arguments,
- headers used by a system call,
- implementation of the system call check routine,
- custom types and function definitions.

The configuration process is demonstrated using the example configuration shown in Algo. 9. Each configuration prototype begins with a comment block defining important attributes according to Tab. 3 used by the configuration parser. The header is located right below the comment block of the wrapper function. Its name must be different from that of the system call and the return value should be *`void`*. The arguments must be equal to the prototype of the system call. The wrapper function body then contains the key logic. This can be a simple marker defining the location where the automatically generated rules or more complex tasks like a custom implementation for emulating the system call are inserted. For the latter case, the different helper macros listed in Tab. 4 are provided to modify arguments without worrying about the data types and the implications of different pointer-based argument types as described in Sec. 3.1.2.

To further clarify the use of the different attributes and helper macros, three example functions are explained in detail. First, there is the open system call defined in lines 1-17. The original system call name used by the Kernel is `SYS_open`. The necessary headers to use the system call with their custom parameter types and flag variables are *"stdlib.h"* and *"stdio.h"*. In line 5, the file name argument is added to the path group. If a global rule is defined to check a path parameter later, this argument will be used. In the next line, the reserved group `permission_flag` is used for the flag's argument. This special group is used by the configuration builder so that file permissions can be validated as shown demonstrated in Algo. 10. Line 8 describes an important

and necessary attribute for this specific system call. It implements that the length of the pointer-based argument called file name has the size of `strlen+1`. With this information, the framework can notice that the argument is a string, and in case of modifications, its buffer length is limited by the length of the string plus one. The plus one is necessary, because the C function `strlen` returns the length of a string without the end marker *"\0"*. Lines 10-16 then define the main logic. At the beginning, a debug section is defined. If the debug flag is set in the rule configuration file, the system call is printed to the system log along with its arguments. This can help to comprehend information about opened files during the application's lifetime. The macro `CHECK_RULES()` at the end defines the location where the generated rules should be inserted according to the rule configuration file. In conclusion, the behavior of the system call depends entirely on the defined rule set.

Attribute	**Description**
systemcall	Defines the name of the system call which the prototype represents.
headers	Speciefies a list of headers needed to use the system call.
set_group[{argument}]	Defines additional names for an argument which can be used in the rule configuration file. Group names are necessary if global rules are defined. The name permission_flag is reserved for arguments defining access rights.
set_length[{argument}]	Informs the configuration parser about the size of the specific system call argument. This attribute is necessary for pointer arguments of variable length. Especially if the size depends on the return value of a system call or another parameter. If no length is specified, the C function sizeof is used.
read_length[{argument}]	Defines how many bytes of an argument should be read from the child application. The value can differ from the argument set_length.
link_update[{argument}]	Enables automatic value updates if the defined argument is modified by the specific rule. Currently only supported for string-based buffer manipulations.

Tab. 3: C configuration file comment attributes.

Lines 19-29 intercept the system call `SYS_read`, which is used to read data from a file descriptor in the after mode described by the annotation `:after`, behind the system call name. This mode specifies that this wrapper function is called after the system call is executed. Intercepting the system call at this stage offers the possibility to check and modify the data read by the system call. To be able to achieve such manipulations by using a simple rule format, the attribute definitions from lines 20-25 are mandatory. `set_lenght` specifies that the maximum length of the buffer created by the calling application is limited to the number of bytes defined in the system call parameter `count`. This information makes sure that the framework will never write to the forbidden memory regions. `read_length` regulates how many bytes should be read from this buffer. For the system call `read`, the size is determined by the return value of the system call. A too large value (depending on the stack size) might result in old data from the initial buffer to be taken into account. Consequently, the attribute `link_update` must also be specified. When the rule-based modifications are applied to the data read by the system call, the return value is automatically updated. If this would not be the case, changing the data from *"ABCDE1234"* to *"AB12"* would result in a correctly modified buffer, but the calling application would get the information that it must read nine bytes instead of four, leading it to read *"AB12E1234"*. Since `read` is a system call modifying the memory space of the calling application, the related argument must be marked with the macro `__OUT`. Otherwise, the framework would create a new space with the new data instead of overwriting the location defined by the caller.

The system call `getcwd` in lines 31-47 is used to retrieve the current working directory of a process. The shown implementation emulates the system call. Therefore, the original system call will never be called. The tracer retrieves the data and returns it instead. In lines 38-40, a function is called that retrieves the working directory of a process, using the information from

Macro	Description
__PID	Id of the process / thread attempting to perform the system call.
__OUT	Defines a pointer based system call argument as a return parameter.
SKIP_SYSTEMCALL()	Disables the execution of the system call.
CHECK_RULES()	Specifies the location where the generated rules are inserted.
DEBUG_BEGIN()	Defines the start of a debug region. The code within the region will only be included if the debug flag is set.
DEBUG_END()	Marks the end of the debug region.
LOG_INFO(char*)	Appends a specific string to the syslog file with the severity level info.
LOG_DEBUG(char*)	Appends a specific string to the syslog file with the severity level debug.
LOG_ALERT(char*)	Appends a specific string to the syslog file with the severity level alert.
LOG_CRIT(char*)	Appends a specific string to the syslog file with the severity level critical.
OVERWRITE(target, source)	Overwrites data of a parameter defined in target with data from the source. Target and source mus have the same data type. If return is specified as target, the return value will be overwritten.

Tab. 4: C configuration file helper macros.

"/proc/(pid)/cwd". After line 40, the argument parameter *buf* stores the working directory of the application. To overwrite the data in the memory space of the child application, the `OVERWRITE` macro is called. Because the system call returns the length of the working directory, line 43 overwrites the return value of the emulated system call with the corresponding length. If the function would terminate at this stage, all of the actions would have no effect because the system call would be carried out normally and overwrite the data. To prevent the system call from being executed, `SKIP_SYSTEMCALL()` must be called. For the last step, it is still possible to apply rule checks. Note that the macro `CHECK_RULES()` should occur in every wrapper function since it will also clean up the memory space used for the pointer-based arguments.

3.3.2 Security Rules

The framework provides an extensive language for defining the access rules for system calls with their arguments taken into account. The same syntax can also be used to define the rules that modify the parameters of a system call. The example configuration shown in Algo. 10 demonstrates some of the possibilities offered by the developed framework.

The configuration file is separated into different sections by square brackets. The section `General` determines the basic system call access rules for the tracer and its tracee. `default_action` defines what action will be performed if a system call is made for which no rule exists. The possible derived values are *terminate*, *skip*, *allow* and *trap*:

- *terminate*:
  causes the termination of the tracee invoking the system call.
- *skip*:
  enforces the kernel not to execute the system call. The error number in `errno` will be set to *ENOSYS*.
- *allow*:
  permits the execution of the system call.
- *trap*:
  terminates the application and prints information about the responsible system call. This option makes more sense for debugging purposes on the tracer side.

A default action on the tracer side can be specified with the option `default_action_tracer`. Using the option scheme `syscall <action>` as shown in line 6 and 7, the access rights to system calls as a whole can be specified. Unlike in the default actions, *trap* is invalid and *modify* can

```
/*
* systemcall:           SYS_open
* headers:              stdlib.h, stdio.h
*
* set_group[filename]:  path
* set_group[flags]:     permission_flag
* set_length[filename]: strlen+1
*/
void sec_open(char *filename, int flags, mode_t mode){
  DEBUG_BEGIN()
    char log[1024];
    sprintf(log, "Process called open(%s, %d, %d)", filename, flags, mode);
    LOG_INFO(log)
  DEBUG_END()

  CHECK_RULES()
}

/*
* systemcall:         SYS_read:after
* headers:            unistd.h
*
* set_length[buf]:    count
* read_length[buf]:   return
* link_update[buf]:   return=strlen+1
*/
void sec_read_after(int fd, __OUT void *buf, size_t count){
  CHECK_RULES()
}

/*
* systemcall:         SYS_getcwd
*
* set_group[buf]:     path
* set_length[buf]:    size
*/
void sec_getcwd(__OUT char *buf, unsigned long size){
  char *cwd = getPidCwd(pid);
  strncpy(buf, cwd, size);
  free(cwd);

  OVERWRITE(buf, buf)
  OVERWRITE(return, strlen(buf))
  SKIP_SYSTEMCALL()

  CHECK_RULES()
}
```

Algo. 9: C configuration file example.

be used instead. This option forces `Seccomp` to always contact the tracer for checking and deciding on the final action. On the client side, the minimal system calls needed by an application are `exit` or `exit_group`, depending on the standard C library used. On the tracer side, the particular system call actions can be defined, using the option scheme `tracer <action>`. For the tracer to be operable, more privileges such as `ptrace`, `wait4` and others are required. The last option `debug` specifies weather the framework is used in a debug mode or not. If activated, all `Seccomp` actions are redirected to the tracer, which then logs information about the system call before the initial action is executed.

The section `Global` allows defining detailed rules that affect all system calls, using arguments with the same group specifier as noted in Tab. 3. The rule in line 16 shows that when accessing

a path containing the string *"/invalid"*, the path is modified and the occurence is replaced with *"/valid"*. An application accessing the file *"/home/user/invalid/config.ini"* will be redirected to *"/home/user/valid/config.ini"*.

Other than the global sections, the rules for the specific system calls can be defined by creating a section with the same name. The section `open` defines the rules for the equally named system call `open`. Each function of the specific section requires the attribute `default` defining the standard action for the system call. The first rule in line 20 determines that the files in the *"config"* folder of the current working directory can only be accessed with the read-only mode. It is, therefore, not possible to create or write files in it. The subdirectory *"create_ write_ only"*, on the other hand, allows files to be created and written in it. For the termination action, based on read permissions, two rules are defined. The first rule specifies that the application terminates when it tries to read data from the sub directory *"logs"*. The second rule dictates that the application terminates if read attempts are made to for the files in the *"/bin"* folder. The last rule in line 24 shows a scenario where all open attempts for files with the extension *".txt"* are redirected to the versions with the extension *".dat"*.

In addition, more complex rules can be established as shown in lines 28 and 29 for the system call `setrlimit`. The function allows specifying two kinds of limits on accessing single resources. The first is a so-called soft limit stored inside the attribute `rlimit_cur` of the `limit`, which is the name of the system call parameter. The stricter hard limit is stored in the attribute `rlim_max`. Rule 28 checks if the application tries to modify the limit `RLIMIT_CPU`, which regulates the number of seconds an application is allowed to use the CPU. If this is the case, a comparison is made between the value that the application is trying to set for the hard limit and the current hard limit of the application. So, if the application intends to increase the CPU time, the system call will fail. In contrast, if it reduces the time, the system call will be executed successfully. In conclusion, the application can only reduce the amount of time it can consume. The second rule in line 29 resolves the behavior for the system call to fail, for some resources, if the soft limit is greater than the hard limit. If this is the case, the soft limit is reduced to the same value as the hard limit.

`chdir` allows changing the working directory of an application. To prevent an application from changing its main directory to the root folder *"/"* or any other sensitive directory, a rule as shown in line 33 can be deployed. This rule basically ensures that the tracee cannot be moved up within the folder hierarchy of its current working directory. It can only go deeper. This can drastically increase the security when combined with a similar rule for the open system call, thus allowing the files to be opened within the current working directory *"./"* only.

The rule in line 37 shows the limited capability of the possible checks concerning file descriptors, which is explained in Sec. 3.2. In case the descriptor path is evaluated and the creation of a copy points to a file in the subdirectory *"secret"* of the current working directory, the rule will prevent the creation of a file descriptor copy.

The framework not only has the ability to work with system calls before their execution, but it also specifies the rules after they have been executed. Such a section must be marked with the attribute `:after` behind the name of the system call, as shown in line 48. The rule basically performs two actions. If the read data starts with *"not"*, it is replaced with *"its"*. The second rule replaces all the occurrences of double spaces with single spaces.

In the last example, a buffer manipulation is performed. Regardless of the location where the application attempts to write data, it gets modified. The six rules in line 55 transform all output into leetspeak, also known as 1337. If an application runs the print command for the text *"This is leetspeak"*, *"†h!5 !5 |33†5p34k"* will appear instead. The example shows that all of the redirection

rules to modify data are executed consecutively.

```
[General]
debug:                  False

# Client specific rules
default_action:         terminate
syscall allow:          exit, exit_group, close, fstat, getrlimit
syscall modify:         gettimeofday, getcwd

# Tracer specific rules
default_action_tracer:  terminate
tracer allow:           ptrace, wait4, getpid, socket, sendto, read, chdir, getcwd,
                        fstat, lseek, lstat, open, close, kill, exit, exit_group,
                        write, readlink, connect, prlimit64

[Global]
path redirect:          dir_contains("/invalid") => "/valid"

[open]
default:                skip
path allow(r):          dir_starts_with("./config")
allow(cw):              filename dir_starts_with("./create_write_only")
terminate(r):           filename dir_starts_with("./logs"),
                        filename dir_starts_with("/bin")
redirect:               path dir_ends_with(".dat") => ".txt"

[setrlimit]
default:    allow
skip:       resource == RLIMIT_CPU && limit->rlim_max > getHardLimit(pid, resource)
redirect:   limit->rlim_cur > limit->rlim_max: limit->rlim_cur => limit->rlim_max

[chdir]
default:          allow
path skip:        not dir_starts_with("./")

[dup]:
default:          allow
fd skip:          fd_path_starts_with("./secret")

[socket]
default:          terminate
allow:            domain == AF_UNIX && type == SOCK_STREAM,
                  domain == AF_LOCAL && type == SOCK_STREAM

[read:after]
default:          allow
redirect:         buf starts_with("not") => "its",
                  buf contains("  ") => " "

[write]
default:          allow
buf redirect:     contains("i")=>"!", contains("e")=>"3", contains("a")=>"4",
                  contains("l")=>"|", contains("t")=>"†", contains("s")=>"5"
```

Algo. 10: Rule configuration file example.

3.4 Configuration Parser

To integrate the configuration files into the final application, a Python script must be used. This configuration builder script takes the rule configuration and the C configuration file as an input and creates new C files, which then are used by the framework. The extract in Algo. 11 shows the generated `Seccomp` rules for the tracer based on the configuration files in Algo. 9 and Algo. 10. In lines 8-10, it can be seen that the framework optimizes the rules by trying to include as many validation checks as possible in the `Seccomp` part of the application. More complex rules are directly redirected to the tracer. The tracer then evaluates which wrapper function needs to be executed. Based on the same configuration files as before, the Python script creates, beside the mentioned multiplexer logic, the output shown in Algo. 12 and Algo. 13.

At the beginning of each function, the parameters are listed in a *`void`* command. This is necessary because not all parameters may be used within the wrapper function. Without these empty instructions, failures may occur depending on the used compiler flags. In a next step, a struct is initialized that defines the default action for the system call according to the defined rules. In case of `open`, lines 12-33 contain the redirection rules. If a parameter is changed, the local copy of the argument will be changed as well, allowing for consecutive execution. After all changes have been made, the permission rules are added, as demonstrated in lines 35-51. The framework makes sure that only one permission rule applies by arranging them into an `else if` structure. In the last step, the final action will be executed and reserved storage is freed.

The `read` system call basically has the same structure. The only difference is, that the return value of the system call is also updated due to the defined attribute `link_update` in the C configuration file. Note that even if a comparison may fail, the function `executeRuleResult` is always called. The reason for this behavior is, that the function also takes care to remove possible memory allocations by the used data structure. This is the case, for example, if the rule in line 13 applies and the new string is stored in it. For the system call `getcwd`, a custom implementation was made to emulate the system call. All macros used were therefore replaced by appropriate C functions provided by the framework.

```
seccomp_ctx ctx;
ctx = sec_seccomp_init(SCMP_ACT_KILL);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(fcntl), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getrlimit), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(fstat), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
// Add specific allow rules
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
  SCMP_A0(SCMP_CMP_EQ, AF_UNIX),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
sec_seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(socket), 2,
  SCMP_A0(SCMP_CMP_EQ, AF_LOCAL),SCMP_A1(SCMP_CMP_EQ, SOCK_STREAM));
// Add general modify rules
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(dup), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(getcwd), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(gettimeofday
    ), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(chdir), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE | PTRACE_USE_AFTER_ONLY
    ), SCMP_SYS(read), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(write), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(open), 0);
sec_seccomp_rule_add(ctx, SCMP_ACT_TRACE(PTRACE_EXECUTE), SCMP_SYS(setrlimit),
    0);
```

Algo. 11: `Seccomp` rules generated by the configuration builder for the tracee.

```
void sec_open(pid_t pid, char *filename, int flags, mode_t mode){
  (void)pid;
  (void)filename;
  (void)flags;
  (void)mode;

  struct sec_rule_result __rule_action;
  __rule_action.new_value = NULL;
  __rule_action.size = -1;
  __rule_action.action = SEC_ACTION_SKIP;

  {
    struct sec_rule_result new_string = changeStringOnPartMatch(pid, "/invalid"
                                , filename, strlen(filename)+1, "/valid", true);
    if (new_string.action == SEC_ACTION_MODIFY){
      filename = realloc(filename, new_string.size);
      memcpy(filename, new_string.new_value, new_string.size);
      __rule_action.action = SEC_ACTION_ALLOW;
    }

    executeRuleResult(pid, new_string, PAR1, false, -1);
  }
  {
    struct sec_rule_result new_string = changeStringOnEndMatch(pid, ".dat",
                                filename, strlen(filename)+1, ".txt", true);
    if (new_string.action == SEC_ACTION_MODIFY){
      filename = realloc(filename, new_string.size);
      memcpy(filename, new_string.new_value, new_string.size);
      __rule_action.action = SEC_ACTION_ALLOW;
    }

    executeRuleResult(pid, new_string, PAR1, false, -1);
  }

  if(((flags & O_ACCMODE) == O_RDONLY) &&
      (stringMatchesStart(pid, "./config", filename, strlen(filename)+1, true))
      ){
    __rule_action.action = SEC_ACTION_ALLOW;
  }
  else if(((flags & O_ACCMODE) == O_WRONLY || flags & O_CREAT) &&
      (stringMatchesStart(pid, "./create_write_only", filename,
                          strlen(filename)+1, true))){
    __rule_action.action = SEC_ACTION_ALLOW;
  }
  else if(((flags & O_ACCMODE) == O_RDONLY) &&
      (stringMatchesStart(pid, "./logs", filename, strlen(filename)+1, true))){
    __rule_action.action = SEC_ACTION_TERMINATE;
  }
  else if(((flags & O_ACCMODE) == O_RDONLY) &&
      (stringMatchesStart(pid, "/bin", filename, strlen(filename)+1, true))){
    __rule_action.action = SEC_ACTION_TERMINATE;
  }

  executeRuleResult(pid, __rule_action, -1, false, -1);

  // cleanup memory
  free(filename);
}
```

Algo. 12: C emulation source generated by the configuration builder.

```
void sec_read_after(pid_t pid, int fd, void *buf, size_t count){
  (void)pid;
  (void)buf;
  (void)fd;
  (void)count;

  struct sec_rule_result __rule_action;
  __rule_action.new_value = NULL;
  __rule_action.size = -1;
  __rule_action.action = SEC_ACTION_ALLOW;

  {
    struct sec_rule_result new_string = changeStringOnStartMatch(pid, "not",
                                                                 buf, count, "its",
    false);
    if (new_string.action == SEC_ACTION_MODIFY){
      memcpy(buf, new_string.new_value,
             (new_string.size > (int)count) ? (int)count : new_string.size);
      modifyReturnValue(pid, strlen(buf)+1);
      __rule_action.action = SEC_ACTION_SKIP;
    }
    executeRuleResult(pid, new_string, PAR2, true, count);
  }
  {
    struct sec_rule_result new_string = changeStringOnPartMatch(pid, "  ", buf,
                                                                count, " ", false);
    if (new_string.action == SEC_ACTION_MODIFY){
      memcpy(buf, new_string.new_value,
             (new_string.size > (int)count) ? (int)count : new_string.size);
      modifyReturnValue(pid, strlen(buf)+1);
      __rule_action.action = SEC_ACTION_SKIP;
    }
    executeRuleResult(pid, new_string, PAR2, true, count);
  }


  executeRuleResult(pid, __rule_action, -1, false, -1);

  // cleanup memory
  free(buf);
}

void sec_getcwd(pid_t pid, char *buf, unsigned long size){
  (void)pid;
  (void)buf;
  (void)size;

  char *cwd = getPidCwd(pid);
  strncpy(buf, cwd, size);
  free(cwd);

  modifyReturnParameter(pid, PAR1, buf, size);
  modifyReturnValue(pid, strlen(buf));
  invalidateSystemcall(pid);

  // cleanup memory
  free(buf);
}
```

Algo. 13: C emulation source generated by the configuration builder.

3.5 Integration

The entire framework consists of seven C code and header files, six Python scripts, and three configuration files according to Tab. 5. Two of the C source and header files called *"sec_seccomp_rules"* and *"sec_syscall_emulator"* are not included in the framework because they are generated automatically by the configuration builder *"SecConfigBuilder.py"* based on all three configuration files.

File	Description	Module
seclib.c seclib.h	Entry function to initialize the framework.	Framework
sec_client.c sec_client.h	Initializes the error handling and `Seccomp` rules for the tracee.	Framework
sec_ptrace_lib.c sec_ptrace_lib.h	Main function library of the framework.	Framework
sec_seccomp_bpf_generator.c sec_seccomp_bpf_generator.h	Module to transform abstract seccomp rules into a Berkeley packet filter.	Framework
sec_tracer.c sec_tracer.h	Tracer component of the framework.	Framework
sec_seccomp_rules.c sec_seccomp_rules.h	Files generated by the configuration builder. Initializes the `Seccomp` rules for the tracer and tracee.	Framework
sec_syscall_emulator.c sec_syscall_emulator.h	Files generated by the configuration builder. System call check and modification source not supported by `Seccomp`.	Framework
SecConfigBuilder.py	Configuration file parser.	Generator
SecCParser.py	Python module to parse the C-based configuration file.	Generator
SecCWriter.py	Python module to export formatted C code.	Generator
SecInfParser.py	Python module to parse the rule configuration file.	Generator
SecRegexBuilder.py	Python module to simplify the creation of regulare expressions.	Generator
SecRules.py	Python module that provides helper structures for the parse process.	Generator
sec_rules.ini	Specifies the access and modification rules for system calls.	Configuration
sec_syscalls_conf.c	Specifies the layout and properties of system calls.	Configuration
source_templates.ini	Source templates used by the configuration builder.	Configuration

Tab. 5: Framework source and configuration files.

The C application in Algo. 14 shows how the framework is integrated into an application. First, the main header file called *"seclib.h"* must be included. To start the framework, the function `run_seccomp_framework(...)` must be invoked as demonstrated in line 9. Calling to this function should be the first and last action of `main` because its contents are executed by the tracer and tracee. The first two arguments of `run_seccomp_framework` are equal to the standard arguments of the standard C `main` function. `argc` describes the number of arguments with which the application is called. `argv` is a two-dimensional char array that contains the arguments. The third and fourth arguments are pointers to a function with the same signature as `main`. `sec_main_before` refers to the function that is called before the security measurements of the framework are applied to the application. The second function pointer called `sec_main_after` refers to a function that is secured by the framework. The differentiation between these two execution states gives an application the opportunity to invoke critical system calls before they are disallowed in the unprivileged section. `sec_main_after` should, therefore, contain the main application source that also parses input from untrusted sources.

After the main function is modified and the library is included, the rules must be configured. The first step is to modify the configuration file *"sec_rules.ini"* to specify the access rules for system calls. In a second step, the signatures for each system call for which parameter checks and manipulations are defined must be configured in the file *"sec_syscalls_conf.c"*. Finally, the Python script *"SecConfigBuilder.py"* can be used to parse these files and generate the missing source files. The python script provides the optional parameters listed below:

- *--conf*:
  path to the system call configuration file.
  Default value = *"sec_syscalls_conf.c"*

- *--rule*:
  path to the system call rule file.
  Default value = *"sec_rules.ini"*

- *--tmpl*:
  source templates for the builder script.
  Default value = *"source_templates.ini"*

- *-o*:
  target folder for the generated files.
  Default value = *"out"*

To compile the application without having to modify references, the generated files must be stored in the same directory as the other C source and header files. Once the steps previously described are performed and the application has been compiled and linked to the files of the framework, the application can be started without the need for any arguments for the executable.

```c
#include <stdio.h>
#include <stdlib.h>
#include "seclib.h"

int sec_main_before(int argc, char **argv);
int sec_main_after(int argc, char **argv);

int main(int argc, char **argv){
  return run_seccomp_framework(argc, argv, sec_main_before, sec_main_after);
}

int sec_main_before(int argc, char **argv){
  (void)argc;
  (void)argv;

  printf("Run privileged operations.\n");

  return EXIT_SUCCESS;
}

int sec_main_after(int argc, char **argv){
  (void)argc;
  (void)argv;

  printf("Run unprivileged operations.\n");

  return EXIT_SUCCESS;
}
```

Algo. 14: Framework integration into a minimal application.

4 Case Study - Nginx

The following chapter demonstrates the usage and possibilities of the proposed framework by integrating it into Nginx a popular web server used by 63% of the world's busiest sites [19]. In addition to running websites, it can also be used as a reverse proxy, load balancer and HTTP cache. Security for these kind of tools is an absolute necessity since their connection is exposed directly to the internet. Hosting dynamic web pages based on languages like PHP further increases the attack surface due to the interaction with sources from third-parties. On the basis of different attack scenarios, common known vulnerabilities and exposures listed on www.cvedetails.com, configurations are built and their effectiveness evaluated [20].

4.1 Integration

To integrate the framework into Nginx, the following steps are required:

1. Download the repository of Nginx from https://github.com/nginx/nginx.
2. Download the framework from https://github.com/deforation/sec_seccomp_framwork
3. Let Nginx build the make files by running *"./configure"*.
4. Place the folders *"Generator"* and *"seccomp_framework"* into *"/src/core"*.
5. Modify the main file called *"nginx.c"* by:
   - Add an include annotation for the file *"seccomp_framework/seclib.h"*
   - Rename the main function to `sec_main_after`.
   - Create a new main function and call
     `return run_seccomp_framework(argc, argv, NULL, sec_main_after);`
6. Add the files of the framework to the make file so they are compiled together with the main application:
   - *"seccomp_framework/seclib.h"*
   - *"seccomp_framework/sec_client.h"*
   - *"seccomp_framework/sec_ptrace_lib.h"*
   - *"seccomp_framework/sec_seccomp_bpf_generator.h"*
   - *"seccomp_framework/sec_seccomp_rules.h"*
   - *"seccomp_framework/sec_syscall_emulator.h"*
   - *"seccomp_framework/sec_tracer.h"*
7. Formulate security rules within the file *"sec_rules.ini"*.
   Additional modifications to the file *"sec_syscalls_conf.c"* may be required.
8. Run the configuration builder of the framework with the command
   "python3 SecConfigBuilder.py -o ../seccomp_framework"
9. Compile Nginx and run it.

4.2 Attack Scenarios

Create and Overwrite Arbitrary Files: CVE-2009-3898

In 2009, the vulnerability CVE-2009-3898 was uncovered that allowed remote authenticated users the create and overwrite arbitrary files. The attacker only needed to change the header of a HTTP-request and change its destination to a path that contains *"../"*. To mitigate the vulnerability, multiple methods exist: access to unused files can be prevented by performing sanity checks on the `open` system call or, alternatively, it can be checked if the request string contains *"../"*. The latter method became the public available fix.

The configuration file extract in Algo. 15 contains both variants of the fix for the vulnerability. The first fix in lines 1-8 minimizes the access permissions for files. Per default, the execution of the system call `open` will be skipped. Unless the application reads its configuration file or files that belong to the web page. Write permissions are only given to log files and the application status file called *"nginx.pid"*. In addition, `Nginx` is required to have permissions to create and read the status file. Note that the security rules use relative paths. The system call `chdir` to change the working directory, should, be disallowed or absolute paths should be considered instead. The second fix is visible in lines 10-13. Each request to the web server will be scanned for the occurrence of the string *"../"* which is then replaced with an empty one. Another solution would be to replace the return value of the system call with *"-1"*. Nginx then thinks that the system call failed and the vulnerable source is invoked.

```
[General]
syscall skip:           chdir

[open]
default:                skip
filename allow(r):      dir_starts_with("./html"),  dir_starts_with("./conf")
filename allow(w):      dir_starts_with("./logs"),  dir_starts_with("./access.log")
filename allow(rwc):    dir_starts_with("./logs/nginx.pid")

[recvfrom:after]
default:                allow
buf redirect:           contains("../") => ""
#redirect:              buf contains("../"): return => -1
```

Algo. 15: Framework fix for `Nginx` vulnerability CVE-2009-3898.

Remote Code Execution: CVE-2009-2629

CVE-2009-2629 is one of the most serious vulnerabilities `Nginx` has had in its history. Attackers could execute arbitrary code through specially crafted HTTP-requests. In certain cases, this led to a buffer underflow and data of the uniform resource identifier (URI) was written to the heap before the allocated buffer. Such bugs cannot be detected on the system call level and prevention using the framework is not possible. The potential damage caused by exploitation, on the other hand, can be reduced. The security rules should therefore be configured in such a way, that only access to a minimal set of required system calls is granted. In case of `Nginx`, access to the following system calls is needed to host html pages: `accept4`, `bind`, `brk`, `clone`, `close`, `connect`, `dup2`, `epoll_create`, `epoll_ctl`, `epoll_wait`, `eventfd2`, `exit`, `exit_group`, `fcntl`, `fstat`, `geteuid`, `getdents`, `getpid`, `getrlimit`, `ioctl`, `listen`, `lseek`, `lstat`, `mkdir`, `mmap`, `prctl`, `pread64`, `pwrite64`, `read`, `recvfrom`, `rt_sigaction`, `rt_sigprocmask`, `rt_sigsuspend`, `sendfile`, `set_robust_list`, `setrlimit`, `setsockopt`, `shutdown`, `socket`, `socketpair`, `stat`, `uname`, `write` and `writev`.

To further reduce the attack surface, access to these system calls should only be granted if the arguments fulfill certain criteria. The system call `setrlimit`, for example, should only be allowed if the first argument equals `RLIMIT_NOFILE` because all other resource limits are never changed by `Nginx`. `mkdir` to create directories should also be restricted to one specific directory. `Nginx` uses this command to create five folders for temporary files.

Regarding this specific scenario, an attacker confronted with such a limited set of system calls will have a hard time to seriously damage the system or steal confidential information. Especially if validation checks to the most dangerous system calls `mkdir`, `open`, `socket`, `ioctl`, `brk`, `dup2`, `setrlimit` and `clone` are applied. According to Massimo et al., these system calls may have the potential to be used for denial of service attacks [21]. `open` can, in some cases, give an attacker the ability to gain full control over the system.

Gain Root Privileges via a Symlink Attack: CVE-2016-1247

In 2016, a vulnerability was published that allowed local users with access to the web server user *"www-data"* to gain root access on the system. All an attacker had to do, is to replace the error log file of `Nginx` by a symbolic link pointing to the file *"/etc/ld.so.preload"*. Like `LD_PRELOAD`, this file is used to overwrite functions of dynamic linked libraries. Contrary to the per process solution `LD_PRELOAD`, all binaries on a system are affected. In general, a normal user does not possess the access rights to create files in *"/etc"*. The module logrotate of `Nginx` runs as root and periodically replaces log files. If this happens, the file gets created with access permissions for the user *"www-data"*. An attacker can then remove the symbolic link and write custom wrapper function to this specific file. Thus allowing to install a backdoor terminal with root access once the wrapper function is triggered. In order to mitigate this attack scenario, it is necessary to know how an attacker can gain access to the user *"www-data"*. Since hosted pages and `Nginx` run under this user, a vulnerability with the ability to execute code is sufficient.

The example configuration in Algo. 15 shows a possibility to weaken the security problem. The first fix in line 2 revokes permission to the system call `symlink`. Without access to this function, an attacker is unable to crate a symbolic link to *"/etc/ld.so.preload"*. If the function call should be allowed, the fix from lines 4-6 can be used. The validation check tests if the target of the symbolic link starts with *"/etc"*. If this is the case, the system call wont be executed. An alternative approach considers the solution offered by lines 8-10. In order to write vulnerable content to the file, the system call `write` has to be invoked. It would therefore be possible to scan each string for important words required by C applications and disallow them. Special attention is required, because an attacker may write the content character wise to the file. In this case, the argument of write can be scanned for brackets. If those are disallowed or replaced by other symbols, no compilable functions can be written.

```
[General]
syscall skip:     symlink

[symlink]
default:          allow
skip:             path1 dir_starts_with("/etc")

[write]
default:          allow
buf skip:         contains("define"), contains("include"), contains("{")
```

Algo. 16: Framework fix for `Nginx` vulnerability CVE-2016-1247.

Obtain Information About the Web Server configuration: Custom scenario

In the example configuration to mitigate the vulnerability with the name CVE-2009-3898, `open` is restricted to have access to a specific subset of files and folders used by `Nginx`. One rule, for example, specifies that the application has permissions to read files from the directory *"./conf"*, in which the configuration files are stored. This means that an attacker may also be able to access this particular set of files. To solve this problem and keep the configuration files secure from unauthorized intruders, `Nginx` must be modified slightly. As shown in Algo. 14, the framework gives the ability to execute code before and after the security measures are applied. Therefore, the configuration files should be opened in the function `sec_main_before` according to the example source in Algo 14. Read and parse of the configuration files can still remain at the same location where the security measures are active. Therefore, the rule for granting permission to open files in the configuration directory can be removed. An attacker would then have to scan the memory of the application to access this information which is unlikely because modern operating systems implement address space layout randomization. This means, that address ranges can hardly be foreseen and thus finding the relevant information in the memory is unlikely.

4.3 Performance

The impact on performance of the developed framework is measured by sending as many requests as possible to the web server. `Nginx` is, therefore, used to host a small site consisting of static content and the functionality to traverse directories using the integrated autoindex module. The number of requests that `Nginx` can process with and without the framework applied to it is measured by the tool `httperf` [22]. `httperf` allows opening multiple connections to a web server and accessing the resources it host while logging all access times.

Sending 700,000 requests to 7 resources along 100,000 sessions took `Nginx` 50.66s to respond. It was, therefore, able to process up to 13,816.6 requests per second. The same scenario on `Nginx` protected by the framework using the rule set defined in Algo. 17, took 109.32s, so that 6403.2 requests per seconds were processed. In conclusion, the performance overhead is 115.78%, which can be neglected considering the gains in security by the drastically reduced attack surface.

```
[General]
debug:                 False

default_action:        terminate
syscall skip:          chdir, chroot
syscall allow:         exit, exit_group, close, fstat, lstat, lseek, getpid, pread64,
                       epoll_create, brk, geteuid, mkdir, ioctl, getrlimit, setrlimit,
                       stat, fcntl, setsockopt, listen, mmap, rt_sigprocmask, accept4,
                       rt_sigaction, clone, pwrite64, dup2, socketpair, rt_sigsuspend,
                       set_robust_list, prctl, eventfd2, epoll_ctl, epoll_wait, bind,
                       sendfile, shutdown, writev, write, read, uname, socket, connect

default_action_tracer:  terminate
tracer allow:           ptrace, wait4, getpid, sendto, chdir, getcwd, lseek, lstat,
                        readlink, kill, connect, fstat
[open]
default:                skip
filename allow(r):      dir_starts_with("./html"), dir_starts_with("./proc/stat"),
                        dir_starts_with("/conf"),  dir_starts_with("/proc/cpuinfo"),
                        dir_starts_with("/etc/localtime"),
                        dir_starts_with("/sys/devices/system/cpu/online"),

filename allow(w):      dir_starts_with("./logs"), dir_starts_with("./access.log")
filename allow(rwc):    dir_starts_with("./logs/nginx.pid")
filename redirect:      dir_ends_with(".txt") => ".dat"

[getdents]
default:                allow
fd skip:                fd_path_contains("nolist")

[recvfrom:after]
default:         allow
redirect:        buf contains("../"): return => -1,
                 buf starts_with("GET /data/private/") => "GET /data/fake_private/"
```

Algo. 17: `Nginx` case study: rule set for performance measurements.

5 Discussion and Future Prospects

The initial task was to build a tool-chain that would make it easier for developers to integrate the aspect of least privileges into their own programs. Because there is no exact definition of the term *"easier"* in this context, the integration process is compared to existing solutions to evaluate if the task could be accomplished successfully.

Two of the four analyzed existing solutions Privman and Privtrans shortly described in Chap. 1, have a similar integration process. In both cases, a developer needs to modify the existing source in multiple locations by replacing function calls or adding annotations to it. ProgramCutter and SPL|T perform both either static or dynamic code analysis to split an application into privileged and unprivileged parts. In some cases, this process may fail if the size of pointer arguments is not known without taking the man pages into account. Furthermore, validation routines must be manually implemented for each of the generated parts. To integrate the proposed solution, on the other hand, a single line of code is sufficient. Also, validation checks and even the sanitization of system calls and their arguments can be defined using a comprehensive configuration language without the need to code C routines. Therefore, the integration process takes much less steps and the solution does not rely on the completeness of any kind of execution traces. As a result, the proposed framework exhibits all requested merits.

Another notable difference from other solutions, in general, is that validation checks can be enforced on the kernel level without the need to actually modify the kernel. The example integration into `Nginx` has further shown that also its effectiveness in terms of security is given for real-world applications.

Although the framework provided offers many functionalities, there are still a number of optional open tasks that could not be solved or integrated in the time given. Future work could, therefore, tackle the problem that currently only Linux kernel versions greater or equal than version 4.8 are supported. This is related to changes made in the order of events in `PTrace`. The abstraction layer for generating BPF-programs can be optimized for larger filter programs by adding a binary search for system calls. The number of checks possible on the kernel level can be increased by also supporting disjunctive validation terms. In any case, there is enough room to extend the number of helper functions that the framework comes with to build more powerful rules. To increase the performance for multithreaded applications, a copy of the tracer could be made at runtime so that each thread belongs to one specific instance of the tracer. Although this would offer the capability to perform multiple system call checks at the same time, new security risks are introduced to which new solutions have to be found. One example of such a security risk concerns the problem where a secondary thread may change the content of a system call argument after the validation check is passed.

6 Conclusion

In this work, a novel framework for the automatic dropping and separation of privileges has been presented. Access rules to system calls are enforced by the Linux kernel. In all other cases, a request broker is invoked to make the final decisions. A new designed configuration language is used to define access rules to system calls, to create validation checks for its arguments, and instructions to manipulate them.

A sample integration of the framework into `Nginx` has shown that a single line of code is sufficient to apply the concept of least privileges to the web server. Known vulnerabilities from the past could successfully be mitigated and the potential damage, caused by arbitrary code execution, minimized.

The use of standard Linux functions like `Seccomp` and `PTrace` ensures absolute portability across different Linux distributions. Although the framework introduces an average overhead of about 60.3% to system calls, the gains in terms of security outweigh the overhead.

The framework named `sec_seccomp_framework` can be found at
`https://github.com/deforation/sec_seccomp_framwork`.

A Bibliography

[1] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[2] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, 2003.

[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, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[4] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 323–333, Piscataway, NJ, USA, 2013. IEEE Press.

[5] M. Trapp, M. Rossberg, and G. Schaefer. Automatic source code decomposition for privilege separation. In *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, Sept 2016.

[6] Federal trade commission. The equifax data breach, 2017. Available at `https://www.ftc.gov/equifax-data-breach`.

[7] NIST. Cve-2017-5638, March 2018. Available at `https://nvd.nist.gov/vuln/detail/CVE-2017-5638`.

[8] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors, September 2011. Available at `https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf`.

[9] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association.

[10] Derek G. Murray and Steven Hand. Privilege separation made easy: Trusting small libraries not big processes. In *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, pages 40–46, New York, NY, USA, 2008. ACM.

[11] Jun Wang, Xi Xiong, and Peng Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 361–373, Berkeley, CA, USA, 2015. USENIX Association.

[12] Ira Ray Jenkins, Sergey Bratus, Sean Smith, and Maxwell Koo. Reinventing the privilege drop: How principled preservation of programmer intent would prevent security bugs. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, HoTSoS '18, pages 3:1–3:9, New York, NY, USA, 2018. ACM.

[13] Wikimedia Commons. Sequence diagram for openssh privilege separation, 2016. Availabla at `https://commons.wikimedia.org/wiki/File:OpenSSH-Privilege-Separation.svg`.

[14] Munawar Hafiz, Ralph E Johnson, and Raja Af. The security architecture of qmail. In *In PLoP 2004 Proceedings*, 2004.

[15] Jack Edge. A seccomp overview, Sep 2015. Available at `https://lwn.net/Articles/656307`.

[16] Jonathan Corbet. Seccomp and sandboxing, May 2009. Available at `https://lwn.net/Articles/332974`.

[17] *SIGNAL_SAFETY(7) Linux Programmer's Manual*, April 2018.

[18] H.J. Lu1, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface, jan 2018. Availabla at `https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf`.

[19] INC NGINX. nginx, 2018. Available at `https://www.nginx.com/?_ga=2.228374173.103440784.1525847183-1521241523.1523906676`.

[20] MITRE Corporation. Cve details - the ultimate security vulnerability datasource, 2018. Available at `https://www.cvedetails.com/`.

[21] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 174–183, New York, NY, USA, 2000. ACM.

[22] David Mosberger, Martin Arlitt, Ted Bullock, Tai Jin, Stephane Eranian, Richard Carter, Andrew Hately, and Adrian Chadd. httperf, 2018. Available at `https://github.com/httperf/httperf`.

B List of Figures

C List of Tables

D List of Algorithms

E Documents

E.1 Assignment of Tasks

Master Thesis: A Framework and Toolchain for Dropping Privileges

Stephan Neuhaus

2017-09-07

1 Introduction

The term "dropping privileges" means the voluntary abandonment of actions that a process is authorised to perform. After a process has dropped privileges, all actions that require these dropped privileges are then illegal and are flagged as erroneous by an instance that the process cannot influence, such as the operating system.

Linux requires at least some system processes to run with superuser privileges, at least part of the time. The reasons for this are varied, and include opening ports with port numbers less than 1024; accessing users' mailboxes when running on behalf of these users; or accessing log files otherwise inaccessible to the logging process. In a monolithic application, this means that the process needs constantly to be aware at what privilege level it is running, and it needs to drop and re-acquire root privileges constantly. This is obviously error-prone and has led to a litany of security advisories in such applications.

A clean design for dropping privileges, and one that allows dropping privileges even in ordinary (non-superuser) processes is the *request broker*. This is an architecture in which there is a reference monitor that is privileged. All other parts of the program run as threads and drop most privileges, except the privilege of talking to the request broker. Whenever a thread needs something done that exceeds its privilege level, such as opening a file for writing, it asks the request broker to do that for it. There are several techniques to achieve this, one being SECCOMP-BPF.

SECCOMP-BPF uses Berkeley Packet Fiters to formulate a policy about which systen calls are permitted for a thread and which are not. Unfortunately, these policies are next to unreadable, because there is no tooling that would allow one to formulate system call policies in a natural way and then translate them into BPF language. This thesis will solve that problem.

2 Task

This thesis should develop a language and toolchain for SECCOMP-BPF policies. Additionally, it should develop a request broker framework that makes it easy or at any rate easier for developers to use a request broker in their own programs.

1

The work in thesis is divided into several parts, all of which should result in a chapter or series of chapters in the final thesis report:

- Research into the various ways of dropping privileges. The candidate should have a good understanding of what it means to drop privileges and why it is so difficult to do right.
- A study of related work in request broker technology.
- Development of a plan for a request broker architecture and its integration with typical programs.

For a passing grade (4.0), the final work must contain:

- A correct summary of the state of the art in dropping privileges. If possible, both academic and nonacademic references should be used.
- At least one case study of a program that attempts to acquire and drop privileges as needed, and assess the consequences of that strategy on the complexity of the codebase and its susceptibility to security vulnerabilities. (This can be an existing program or a program written for the purpose. If it is the latter, the program cannot be a mere proof-of-concept.)
- A viable plan for a request broker architecture and its realisation with SECCOMP-BPF.
- A viable design for a policy language for SECCOMP-BPF.
- A viable plan for integrating the policy language with the request broker architecture, taking the case study or studies above as an example.

For a nomination for highest grade (6.0), the work must also result in at least one of the following (the more the better):

- Working implementations of all the plans listed above.
- A free (as in freedom, not beer) implementation of the fuzzer that is useful for other protocol developers. (Acceptance as a Debian package a plus.)
- An academic paper, either as main author or as coauthor, submitted to a good conference. (Acceptance a plus, but not required.)

2