FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Straightjacket: Tightening Process Execution Policies at Runtime

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Adel Gadllah BSc
Matrikelnummer 0828330

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Georg Merzdovnik BSc

Wien, 13. April 2015

Adel Gadllah	Edgar Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Straightjacket: Tightening Process Execution Policies at Runtime

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Adel Gadllah BSc
Registration Number 0828330

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Georg Merzdovnik BSc

Vienna, 13th April, 2015

Adel Gadllah	Edgar Weippl

Erklärung zur Verfassung der Arbeit

Adel Gadllah BSc
Am Schöpfwerk 29/14/27 A-1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. April 2015

Adel Gadllah

Acknowledgements

I'd like to thank my advisor Edgar Weippl for allowing me to work on this thesis and the team at the SBA Research Group for providing the infrastructure in form of a git repository and access to the *SPEC CPU2006* benchmark suite. I'd also like to thank Matthias Neugschwandtner for suggesting this interesting topic. Special thanks go to my co-advisor Georg Merzdovnik for his support, valuable feedback and advice during my work on this thesis. Finally, I'd like to thank everyone else who I worked with during my bachelor and master study at the Vienna University of Technology.

Kurzfassung

Das Principle of the Least Privilege ist ein bekanntes und gängiges Prinzip für die Entwicklung von sicheren Anwendungen. Es besagt, dass Anwendungen nicht mehr Privilegien haben sollen als für deren Aufgabe notwendig ist. Das minimiert den Schaden, den ein erfolgreicher Angriff verursachen kann. Bestehende Sandboxing Techniken fokussieren jedoch darauf die Privilegien ganzer Anwendungen zu beschränken. Unterschiedliche Teile einer Anwendung benötigen jedoch unterschiedliche Privilegien. Diese Arbeit stellt Straightjacket vor, eine Sammlung von Werkzeugen für das Betriebssystem Linux, die erlauben die Privilegien von Anwendungen auf der Ebene einzelner Funktionen zu definieren. Entwickler können die Privilegien von Funktionen durch Anmerkungen im Quelltext der Applikation definieren ohne die Funktionen selbst umschreiben zu müssen. Darüber hinaus enthält Straightjacket einen statischen Analyzer, der dabei hilft die benötigten Privilegien einer Funktion zu ermitteln. Veränderungen am Betriebssystem sind nicht notwendig. Die Evaluierung zeigt das Straightjacket gängige Exploits blockiert ohne signifikante Auswirkungen auf die Anwendungsperformance zu haben, wenn die Anmerkungen nicht übermäßig eingesetzt werden. Das wäre der Fall wenn beispielsweise jede einzelne Funktion einer Anwendung mit Anmerkungen versehen wird. Straightjacket wird auch mit Fokus auf die Benutzbarkeit entwickelt und kann ohne großen Aufwand in bestehende Entwicklungsumgebungen und Arbeitsabläufe von Entwicklern integriert werden.

Abstract

The principle of the least privilege is a well-known and established design practice for developing secure applications. It states that applications should not have more privileges than necessary to operate. That limits the damage that can be done by an attacker in case of a successful exploit. Existing sandboxing technologies focus on confining whole applications. However, different parts of an application require different privileges than others. This thesis introduces Straightjacket, a set of tools for the Linux operating system that allow assigning different levels of privileges to individual functions of an application. It allows the developer of the application to define the required privileges by adding annotations to the application's source code without having to rewrite the affected functions. Straightjacket includes a static analyzer that helps to identify the required privileges of a specific function. It does not require any operating system modifications. The evaluation shows that Straightjacket blocks common exploits successfully. The introduced performance impact is not significant as long as the annotations aren't overused. Overusing for instance would be annotating every single function of an application. Straightjacket is developed with usability in mind and can be easily integrated into existing build processes and developer workflows.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Problem statement 1
1.2 Methodology 2
1.3 Overview 3

2 Fundamentals **5**
2.1 Operating system internals 5
2.2 Sandbox 7
2.3 Seccomp 7
2.4 Text processing 7
2.5 Static analyzer 9
2.6 Buffer Overflow 9
2.7 Shell Code 9

3 State-of-the Art **11**
3.1 Sandbox techniques 11
3.2 Performance overhead of sandboxes 14
3.3 Security of sandboxes 15
3.4 Summary and Conclusion 16

4 Annotation based sandboxing **19**
4.1 System overview 19
4.2 Run time Enforcement 21
4.3 Annotation 26
4.4 Code generator 29
4.5 System call identification 32
4.6 Automated annotation 37
4.7 Usage 39

5 Evaluation **41**
5.1 Exploit mitigation 41
5.2 Performance 50

6 Summary and Conclusion **57**

7 Further work **59**

Bibliography **61**

CHAPTER 1

Introduction

This chapter gives an overview of this thesis. It describes the problem that this thesis is solving as well as the used methodology followed by an overview of the thesis structure.

1.1 Problem statement

The principle of the least privilege [1] is a well-known and established design practice for developing secure applications. It states that applications should not have more privileges than necessary to operate. That limits the damage that can be done by an attacker in case of a successful exploit. [2]

There are multiple ways how an operating system can limit an application's privileges. Operating systems implement discretionary access control (DAC), which controls access rights based on ownership of objects like files, directories and processes and/or mandatory access control (MAC), which restricts access unless explicitly granted by a policy. One example of a MAC system on Linux is SELinux. Access control lists (ACLs) are also implemented by modern operating systems to complement traditional file access permissions. [3] [4] [5]

Access controls can be circumvented by attacking the entity that enforces them. For instance to circumvent policies imposed by the Java VM, the attacker has to focus on attacking the Java VM itself. For native applications that entity is the operating system kernel. The entry points from user space to kernel space are system calls. A system call lets the kernel do work on behalf of a user space process based on the input provided by that process. A bug in the processing of the user supplied input may allow an attacker to circumvent kernel enforced privileges. To reduce the likelihood of such an attack the set of available system calls to an application should be restricted to the ones that are actually needed to do the desired job. [6] [7] [8]

In addition to the operating system provided access controls sandboxing systems have been developed to confine and isolate applications from each other while limiting the set of available privileges further. Sandbox solutions are generally used when executing code from untrusted sources. One example is Google's native client [9], which executes native code inside a web browser. Other examples are application sandboxes used on mobile operating systems like Android, Firefox OS or iOS. [10] [11] [12] [13]

Currently available and deployed systems focus on limiting the privileges of a single process or a group of processes. But different parts of an application may require higher privileges than others. For instance some part of the application might need to access file system or network resources, while other parts can operate without any of those. The consequence is that developers have to either accept that some parts of their programs runs with higher privileges than necessary or resort to multiple process solutions that increase the complexity of the software and thus increase the likelihood of adding new vulnerabilities in the code.

Existing sandboxes and MAC systems are also mostly disconnected from the program. For instance the SELinux policy on systems like the Linux distributions Fedora or Redhat Enterprise Linux is in a centralized selinux-policy package which is not written by the application authors.

Having a set of tools that allow developers to restrict privileges by annotating sections in the program code would solve those problems. Privileges are restricted by reducing the set of available system calls and/or system call arguments. The limited set of available system calls would limit the attack surface that could be used to escalate privileges by attacking the kernel and at the same time different parts of the process can run with different set of privileges. In addition to preventing attacks on the operating system kernel the applications themselves are protected from generic shell codes that open a command shell or make network connections to an attacker's machine, by limiting the system calls that an attacker could use. For instance if executing binaries is not allowed, the attacker cannot spawn a command shell. If the system calls required for network access are disallowed the attacker cannot run shell code that connects to the network. By only having to annotate the code, the developers do not have to write code that does the actual enforcement.

1.2 Methodology

The methodological approach consists of three steps.

The first step is a literature review that helps to get background knowledge of the required technologies as well as determining the state of the art in the area of sandboxing technologies.

The second one is the implementation of the tools starting with the run time enforcement code, followed by defining a suitable annotation syntax and developing the code generator that transforms the annotations into run time enforcement code. In addition to developing the code generator a static analyzer to help with system call identification and a tool to automatically apply annotations based on the identified system calls are developed.

Finally, the results are evaluated in terms of exploit mitigation effectiveness and performance overhead. Exploit mitigation effectiveness is evaluated by analyzing the restricted set of system calls available in the context of exploits for a selection of publicly documented vulnerabilities. Performance is measured using the *SPEC CPU2006* benchmark suite. After running the benchmarks unmodified to determine the baseline performance, the benchmark runs are repeated with different levels of annotations to measure the performance overhead.

1.3 Overview

The structure of this thesis follows the steps of the methodology.

It is organized as follows:

In chapter 2 fundamentals required for understanding the following chapters are presented. It is followed by chapter 3 where the state of the art of sandboxing technologies with focus on run time and partial process policies is discussed. Afterwards in chapter 4 the implementation of the proposed solution is described in detail followed by a short usage documentation of the developed tools. In chapter 5 the developed solution is evaluated in terms of exploit mitigation effectiveness and performance impact. In chapter 6 the thesis is summarized and a conclusion is drawn from the results. Finally, based on the results limitations and possible further improvements are discussed in chapter 7.

CHAPTER 2

Fundamentals

This chapter explains some fundamentals that are required to understand the proposed solution as well as the implementation of this thesis. The first section explains operating system internals like kernel, processes, threads and system calls. While the concept of kernel, processes or thread are present in almost all operating systems details here focus on UNIX, or in particular Linux, because the implementation is written for the Linux operating system. In addition to that methods and techniques that are required for the implementation like seccomp, text parsing or static analyzing are covered in this chapter. The implementation and how the concepts introduced in this chapter are used are explained in detail in chapter 4. Finally, fundamentals required for understanding the exploit mitigation evaluation are described.

2.1 Operating system internals

2.1.1 Kernel

The kernel is the central part of an operating system that is responsible for managing resources and user programs. It is the first part of an operating system that gets executed when booting the system. It usually runs with higher privileges then ordinary user processes and is responsible for managing and enforcing privileges. The kernel abstracts hardware, manages the system memory and offers services for user programs to interact with it. Those services can be used by programs for tasks like accessing hardware, communicating with each other, creating new processes or allocating resources. The primary means for programs to interact with the kernel is by issuing system calls. [14]

2.1.2 System call

A system call is a routine provided by the operating system kernel that can be used by user programs to do various tasks like reading from a file, executing a new program or

opening a network socket. That works by setting the values of the system call like its unique number and arguments in processor registers and then trigger an interrupt which signals the kernel that a system call is to be executed. The number of the system call can differ between CPU architectures; the registers for the arguments are specified in the architecture's calling conventions. The kernel then suspends the process and reads the values from the processor registers, performs the desired operation and writes back the result into either processor registers and/or the memory of the calling process and switches back to the process's context. This means that a system call runs within the kernel's context and therefore a bug in the kernel's handling of a particular system call can potentially be exploited to escalate the privileges of the calling process. In addition to that the actions that a process can do without issuing system calls are limited. It cannot create nor interact with other processes and cannot access resources like the file system or the network without using system calls. [15] [14]

2.1.3 Process

A process is a running instance of a program. Once a program is executed the operating system starts a new process for it. The process contains both the program code and its (current) state, also called context. The context is used by the operating system to manage the process and by the processor to execute the program. This includes processor register values like the program counter, which stores the current position in the program that is being executed, meta-information like a unique process id, execution priority or the information about the owner of the process. A multitasking operating system manages multiple processes simultaneously which means that a running process can get suspended to make room for another process to be executed. The context is used by the operating system to save and restore the state of the processes during such a switch, hence this action is called *context switch*. Part of the process is also the volatile memory used by the process as well as resources like file descriptors.

Processes are isolated from each other which mean that two processes cannot access each other's memory. This is important for both system stability and security. Processes can communicate with each other using inter process communication (IPC) facilities provides by the operating system like shared memory, sockets or pipes. [14]

2.1.4 Thread

Threads are similar to processes with the distinction that they belong to a specific process and share both memory and resources with it. Threads are used to do work while the process is being blocked for instance waiting for user input or an I/O operation to finish. Another use case is to split a task in multiple parallel sub tasks, so that each of them can run simultaneously and improve performance by distributing the work on multiple processors or processor cores. This is possible because threads can be scheduled by the operating system like processes. In fact on Linux threads are implemented as processes that share the memory and other resources like file descriptors with the parent process.

This implies that threads can directly manipulate the memory of the process and therefore do not need to use inter process communication for interaction, which lowers the overhead and thus improves performance. This can cause problems when multiple threads try to access and/or manipulate the same memory regions simultaneously. To deal with that problem the operating system and/or the processor provide locking mechanisms and atomic operations. An atomic operation cannot be interrupted and therefore the memory it accesses cannot be changed by a different thread while it is executing. [14]

2.2 Sandbox

A sandbox is a system that allows running code in an environment with a well-defined policy that restricts the privileges of that code. A sandbox can be used for multiple purposes such as running untrusted programs in a safe environment or to limit the damage caused by compromised applications. A program running outside of a sandbox in general has the same privileges as the user that executed it. [16] [17]

2.3 Seccomp

Linux 2.6.23 introduced the secure computing mode, in short seccomp, which allows limiting the system calls available to the calling process. When a process enables this mode, called *SECCOMP_MODE_STRICT*, using the *prctl* system call the process can only use the system calls *read*, *write*, *_exit* and *sigreturn* from this point on. Trying to call any other system call results into the kernel sending a SIGKILL signal to the process and thus forcibly quit it. In Linux 3.5 a second mode, called *SECCOMP_MODE_FILTER* or seccomp-bpf, has been introduced that allows loading a custom Berkeley Packet Filter (BPF) into the kernel which can arbitrarily filter (deny or allow) system calls and system call arguments.

At first the application loads the filter in form of a BPF program into the kernel. From this point on the kernel invokes the BPF program each time the application issues a system call and decides whether to allow or deny the system call based on the result of the BPF program.

The filter mode provides the required flexibility for the implementation of the run time enforcement of the annotated system call privileges. [18]

2.4 Text processing

STRAIGHTJACKET is implemented as an additional compilation step and therefore text processing methods are required for the implementation. Those are described in this section.

2.4.1 Lexer

A lexer or tokenizer is a tool that reads an input file or stream and split it into entities called tokens based on predefined rules. Tokens have a higher level semantics then the input that they got created from. For instance a lexer can be used to read a text file and generate a token for every word. Those tokens can then be used for further processing of the input. Such a program can be generated by a tool like *flex*, which has been used for the implementation in this thesis. [19] [20]

2.4.2 Parser

A parser is a program that reads an input, often in form of tokens, validates its syntax, determines its structure and applies a semantic meaning to it. For instance a parser can be used by a compiler to analyze the source code and generate machine code that matches the semantics of the source code. A parser can be generated using tools like *bison* or *yacc*. For this thesis *bison* has been used, which is described in the next section. [19] [20]

2.4.3 Bison

Bison is a parser generator compatible with *yacc*. It uses an input file ending with the extension .y which contains the grammar and actions that should be executed on specific nodes. The input consists of tokens generated by a lexer. A *bison* rule is basically a grammar rule that consists of a non-terminal on the left hand side and either terminal symbols (tokens) or other non-terminals on the right hand side. Multiple rules on the right hand side can be separated using a pipe (|) operator which basically acts as a logical or. That means that the rule applies to either of the structures on the right hand side connected by the pipe operator. Each of the rules on the right hand side can optionally have C code between curly braces which gets executed once the rule matches a part of the input stream. *Bison* also allows recursive rules. A rule is recursive if the non-terminal does not only appear on the left hand side but also on the right hand side. This is often used to parse lists, like for instance in this thesis for parsing the list of function arguments. One pitfall here is that a rule can be expressed using either right or left recursion. Right recursive means that the non-terminal appears on the right hand side of the rule, while left recursive means that the non-terminal appears on the left hand side of the rule. While both have the same expressive power, i.e. they can match the same inputs, the *bison* documentation recommends to always using left recursion because of how the *bison* parser algorithm works. It uses a stack internally to keep track of the processed terminals and non-terminals. A left recursive rule uses a bounded stack space; while a right recursive rule uses stack space proportional to the size of the parsed input. The latter can cause the stack to fill up quickly for large input files resulting into out of memory errors. [20] [19]

2.5 Static analyzer

A static analyzer is a tool that analyzes a computer program by examining its source code or executable without actually running the program. In contrast to a static analyzer a dynamic analyzer examines the program by running and monitoring it. A static analyzer can be used to gather information about the program's structure like its control flow or call graph. It can also be used to find flaws in the program like memory leaks and security vulnerabilities. For instance Evans at al. have developed a static analyzer that tries to find security issues in applications. [21] It is also possible to use both dynamic and static analyzing, where the results of the static analyzer are used to guide the dynamic analyzer to combine the advantages of both approaches. [11] [22]

2.5.1 LLVM

LLVM is a collection of tools and libraries for building compilers and related tools like static analyzers and debuggers. It provides both its own assembly like language, called *LLVM IR* (Intermediate Representation) and binary representation. *LLVM IR* is in static single assignment (SSA) form in which variables are defined exactly once and each assignment introduces a new version of a variable. This simplifies compiler optimization algorithms. *LLVM* is not only used by the clang compiler, which is part of the *LLVM* project itself, but also by external projects like the mesa graphics library where it is used to compile and optimize shader code for graphic processing units. The *LLVM* project also provides a static analyzer that can be used on its own and is integrated into development environments such as Apple's XCode or the code assistance plugin for the gedit text editor. [23] The static analyzer developed as part of this thesis is based on *LLVM*.

2.6 Buffer Overflow

A buffer overflow is a bug in a program that allows an attacker to overwrite data on the program's stack. This happens when user supplied input is copied into a variable that is stored on the stack without verifying that the data actually fits into it. The stack also contains the return address of the function from where the program's execution continues after the function returns. By overwriting the return address the attacker can hijack the program's execution and let it execute arbitrary code. [24]

2.7 Shell Code

When exploiting a memory corruption vulnerability, like for instance a buffer overflow, an attacker can often run arbitrary injected code with the privileges of the program. The most common payloads try to open a command shell which allows the attacker to run arbitrary commands as if he was logged into the system with the program's privileges. That's why such code is commonly referred to as shell code. For remote exploitable vulnerabilities however a local shell wouldn't help the attacker to gain access to the

system. By opening a remote shell that listens on a network port similar to ssh or telnet the attacker can connect to the victim's system and run arbitrary commands. In case this is not possible, for instance due to a firewall, the attacker can attempt to open a reverse shell, which is a shell that connects back to the attacker's machine to receive commands from there. However shell code is not restricted to opening shells. [24] [25]

CHAPTER 3

State-of-the Art

Research in the area of restricting application privileges goes into multiple directions. Some approaches rely on dynamic instrumentation or system call interception. While others focus on capabilities, a system that allows applications to get privileges that are generally only available to super users (root) without giving them the full set of privileges that a root process would have. [26] [27]

3.1 Sandbox techniques

One of those is *Capsicum*, implemented for the FreeBSD 9 operating system. It adds additional UNIX APIs that allow restricting access to resources like global file system namespaces. For instance it restricts access to the /dev file system which contains device nodes, that allow accessing hardware devices or system memory using regular file operations. In addition to that it restricts some system call parameters to limit the view of the system for the sandboxed application. One such example is the *sysctl* system call for which it only allows around 30 out of 3000 possible parameters. That way a sandboxed application can only use it to query information about its own process but not gather system global information. Other examples of such system calls that act on global objects are *shm_open* and *openat*. Those are restricted in a similar way. The sandbox works by using a library called libcapsicum that a process can use to monitor and maintain a sandboxed child process. Both processes can communicate via a UNIX domain socket. The supervisor process, i.e. the parent process, can change the capabilities of the sandboxed child at run time. They have demonstrated its use by applying it to the Chromium Sandbox and comparing it to existing methods. The system however focuses on confining entire processes. [28]

Capabilities are also implemented on Linux but are far more limited. They can be used allow unprivileged processes to do privileged operations but are not flexible enough

to be a basis for a sandbox implementation. However, they can be used to reduce the attack surface by having applications that otherwise would have to run with root privileges run as unprivileged user. [27]

A solution presented by Shioya et al. implements a sandboxing system which changes its behavior based on the current execution context of a program. The restrictions that are applied by that sandbox therefore are not static over the lifetime of a process. Based on function call chains and a policy file it decides which system calls a particular function may use. In addition to limiting the access to system calls, a more relaxed approach which only allows a system call with specific parameters is also implemented. A function call chain is defined as follows: Suppose the *main* function calls a function named f which in turn calls a function named g. That means that inside the function g the call chain is *g-f-main*. The policy file defines which system calls are allowed in the context of *g-f-main*. The enforcement is implemented using a Linux kernel module that intercepts the system calls. The effectiveness of the approach has been demonstrated by applying it to the Qpopper POP server. The performance impact has been measured to be as low as 4%. [29]

A solution introduced by Liang et al. focuses on file system operations of untrusted programs. It intercepts file write operations and applies them to a cache, called modification cache. The file system visible to the sandboxed application includes those changes. Once the application terminates the user can view the modifications and either commit or discard them. While the primary focus is on file system accesses, the goal of the solution is to prevent any operation that changes the system state. This is achieved by monitoring more system calls than strictly necessary for just file system operations. The system call interception is implemented using the *ptrace* system call, which works like a debugger. This also implies that the sandboxed processes cannot be debugged by a regular debugger, because a process cannot be monitored using *ptrace*[1] from more than process at the same time. [30]

A similar system called Mbox introduced by Taesoo Kim and Nickolai Zeldovich uses seccomp-bpf in addition to *ptrace* to improve the performance. The monitoring process does not have to run for every system call but only for those where the bpf filter triggers a Sigsys signal. However, the solution still imposes a performance overhead between 0.1% and 45.2% and suffers from the same problems due to the usage of *ptrace*. [31]

Kiriansky et al. introduced a solution named *Program Shepherding* which is based on the Dynamo RIO instrumentation framework. It monitors control transfers within a program like jump and return instructions and validates them against a policy. For instance it can restrict access to shared libraries outside of defined entry points. Another check it applies is that it prevents return instructions that do not target an instruction directly after a call instruction. That way overwritten return addresses, a common technique used by exploits to gain control over the program, get detected and denied

[1] man ptrace

execution. In order to avoid the performance impact it uses caching techniques for frequently executed code parts. One major advantage of this approach is that it needs no modifications to existing programs. On the other hand dynamic instrumentation is inherently architecture specific. The presented solution is limited to the x86 architecture. [32]

Bryan Ford and Russ Cox developed a sandboxing library called *Vx32* that works entirely in user space and therefore does not require any kernel modifications. It runs on the operating systems Linux, FreeBSD, and Mac OS X. It works by using dynamic code translation, a technique used primarily by virtual machines to allow a guest operating system to run unmodified on a host processor that has no dedicated virtualization support, to disallow unsafe instructions. It isolates memory accesses using the segmentation hardware supported by the x86 architecture. They also did performance evaluations and concluded that the performance overhead lies between 10% and 50%. While *Vx32* is portable to different operating system due to the lack of kernel modifications, it is tied to the x86 architecture. However, the concepts can be applied to other processor architectures that support the required features. [16]

Rajagopalan et al. introduce an approach called *System call monitoring using authenticated system calls* that uses binary rewriting, cryptography message authentication codes (MAC), and static analyzing to enforce a system call policy. It works by first analyzing the program's binary and finding system call invocations. This is done by searching for the "int 0x80" instruction, which on x86 triggers a system call. The number of the system call is retrieved from writes to the EAX register. Similarly arguments are retrieved from writes to other registers. The MAC is build out of the system call, its position, its static arguments and return address. The binary is then rewritten to pass that MAC as an extra parameter to the kernel. In contrast to solutions that employ security by using signed binaries ([33], [34]) the cryptography is only applied to system calls. The kernel then verifies the MAC at run time before executing the system call. The advantage of this approach is that it works on binaries and does not require manual policy writing and has a low performance overhead. However, it requires that binaries are compiled with relocatable code, but in practice most binaries are not complied with relocatable code and therefore have to be recompiled from source. As with other binary based solutions the analysis as well as the binary rewriting step is inherently architecture specific. [35]

Silver et al. introduce an approach in a thesis called *Implementation and Analysis of Software Based Fault Isolation* which implements address space isolation inside a single process. It works by injecting additional instructions that trigger traps when a code part tries to access memory outside of its assigned region, called *fault domain*. The goal of the work was to avoid the cost of inter process communication while still having separate address spaces. It has been implemented on the DEC Alpha platform. The additional instructions, traps and validations add a performance overhead of around 30% compared

to an unmodified program. [36]

Systrace is a system call interposition mechanism for the OpenBSD operating system. It enforces system call policies on unmodified programs. When a system call not listed in the policy is run it can interactively ask the user for permission. This however relies on the user being able to make correct decisions which is not always the case especially for inexperienced users. [37] Many sandbox technologies build on top of it because it is provided by the operating system. Aleksey Kurchuk and Angelos D. Keromytis introduced an approach to build recursive sandboxes where a sandboxed process uses *systrace* to further confine its child processes. [38]

In addition to securing access to information, sandboxing solutions can also be used to restrict physical system resources like processor time or the amount of memory an untrusted program can use. A malicious program can otherwise attempt a denial of service attack by overusing such resources. A sandboxing solution developed by Dan et al. from the IBM Watson Research Center implements such restrictions in addition to restrictions to logical resources like file or network access. [39]

Sandboxes are also used by malware researchers and anti-virus programs to confine potentially malicious software while analyzing them. [40] System call interception is also used by intrusion detection systems that detect malicious behavior by examining system call usage. [15]

3.2 Performance overhead of sandboxes

Virtual machines may also be used for software isolation, but they come with an overhead especially when dealing with I/O and networking operations compared to simpler solutions that do not virtualize a whole operating system. [41] A survey conducted by Wen et al. compared existing sandbox solutions with virtual machines and concluded that existing sandboxing systems are too restrictive while virtual machines add a high performance overhead. [42] Another survey done by Faisal Al Ameiri and Khaled Salah compared the performance of application level sandboxes and concluded that they mostly affect I/O performance. [43]

To avoid the overhead of virtual machines and thus allow more customer applications to be hosted on the same physical machine, cloud computing providers started using sandboxed containers instead of virtual machines to separate customer instances from each other. [44] [45] A PaaS (Plattform as a Service) system introduced by Krude and Meyer implements such a container framework using seccomp. They conclude that by using seccomp their solution can run on unmodified Linux systems and that there is no performance overhead for computation tasks. Setup costs were also low and the performance impact on network throughput was considered acceptable. [46]

Yoshihiro Oyama et al. proposed a solution to the performance overhead problem added by security checks done by system call interception sandboxes. Their system tries to avoid the expensive checks by predicting the program's behavior based on system call profiles of the program. A separate thread runs security checks for the predicted system calls ahead of time and stores the results. Once the program tries to execute the system call the result is either already available and the access can either be granted or denied or the prediction was wrong in which case the security check has to be run in real time. The system provides speed ups compared to a system that blocks the program each time a security check has to be executed when two conditions are met. At first the security check overhead has to be over a certain threshold and the amount of work a program does between two system calls has to take enough time so that the security checks can run in parallel. [47] A different approach introduced by Bo et al. uses seccomp in combination with PBE theory to distinguish legitimate from malicious system calls. [48]

3.3 Security of sandboxes

Most sandboxes apply restrictions based on system calls, given that those are the primary way of interacting with the operating system. However, when not done correctly such restrictions can be easily circumvented by "creatively" using the available system calls. For instance UNIX systems follow the "everything is a file" principle which means many types of resources are identified and accessed by file descriptors. For instance a security framework or a sandbox could limit access to sockets by monitoring access to the *socket*, *bind*, *read*, and *write* system calls and then decide based on the passed in file descriptor whether the application is allowed to read or write data to the socket. But if it does not monitor the *dup* and *dup2* system calls as well, the file descriptors initially returned by the *socket* system call can be exchanged with others. Which means that further *read* and *write* system calls would entirely bypass the sandbox. Tal Garfinkel has summarized such traps and pitfalls in the paper *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.* [49] Other limitations of system call based sandboxes are that denying a system call can have negative impacts on the behavior of the application and might even lead to other security vulnerabilities. [50] For instance improper handling of concurrency can lead to exploitable vulnerabilities as demonstrated by Robert N. M. Watson. [51] This is especially true for user space based solutions. A malicious application can try to replace the arguments of a system call from another thread shortly after the monitoring process has validated them but before the system call actually gets executed and escape the sandbox that way. This is possible because threads share the same address space. [52]

Tal Garfinkel introduces a system called Ostia that attempts to solve those problems. Instead of the common design of using a monitoring process that intercepts system calls and decides whether to allow or deny them based on a policy, applications delegate operations to agent processes that do the work on behalf of them. It works by using a kernel module that intercepts system calls and calls back into a user space emulation

library which finally delegates the actual work to agent processes. Those do the work on behalf of the application in case it is allowed by the policy. [53]

With the increasing use of sandboxing technologies attackers try to circumvent their restrictions by attacking the kernel. This means that the reduction of the kernel's exposure to applications is a crucial part when developing secure sandboxing systems. Kurmus et al. introduced a system called *kRazor* which applies restrictions based on individual kernel functions and not just system calls. It issues a reachability analysis from system call entry points required by the application and restricts the available kernel functions to only those that are required. They demonstrated that the exposed kernel functions can be reduced by 30% to 80% percent depending on the application. [7] [8]

3.4 Summary and Conclusion

Sandboxing technologies are used in a various areas ranging from mobile devices, web browsers to cloud providers. They are also used for running or analyzing untrusted code like malware. The majority of existing tools however focus on confining entire processes.

Capsicum and the solution presented by Shioya et al. are implemented by modifying the operating system. The former introduces an API that allows a supervisor process to sandbox child processes while the later works with an external policy definition that gets enforced by the kernel based on function call chains. Other approaches such as *Program Shepherding* or the *Vx32* library work entirely in user space and therefore aren't tied to the operating system. In fact *Vx32* runs on multiple operating systems. They do however rely on the underling CPU architecture due to the use of dynamic instrumentation or dynamic translation. While *Vx32* is a library that can be used by modified applications, *Program Shepherding* does not require any program modifications and acts just on the program binary. A different approach by Rajagopalan et al. uses cryptography to authenticate system calls against a policy. It does however require kernel level modifications and the installation step, which does static analyzing on the binary and generates a new binary, is tied to a specific CPU architecture. Other solutions like Mbox make use of user space tracing using the *ptrace* system call.

Sandboxing technologies in general add a performance overhead due to the additional checks that have to be done. A possible solution to this is trying to do the computations ahead of time using a speculative approach where further system calls are predicted from the past behavior. This leads to good results when the application does enough work in user space between two system calls to leave time for the parallel security checks. The prediction also has to be correct. Seccomp based solutions tend to perform relatively good because user space processing, like computation task, are not affected by the sandbox at all.

Sandboxes primarily work by intercepting or monitoring system calls, because they

are the primary means of interacting with the operating system. However, there are some system calls that can be used differently as they were intended and render the sandbox ineffective when not considered as demonstrated by Tal Garfinkel. In addition to that an application might not deal with a denied system call and enter undefined and potentially vulnerable behavior once a system call is denied by a sandbox. User space based solutions are also vulnerable to race conditions while validating system calls.

While there exist sandboxing solutions that can change the enforced privileged at run time they suffer from limitations like being architecture specific, require operating system modifications, introduce a large performance overhead, or requiring large source code modifications.

The solution presented in this thesis, Straightjacket, avoids the problem of undefined behavior after a denied system call by simply killing the process when a violation occurs. It does not use any user space monitoring process which removes a potential attack target and performance overhead source.

CHAPTER 4

Annotation based sandboxing

This chapter describes the annotation based sandboxing approach and issues encountered during the implementation in detail. The code generator processes an annotated C program file and generates a new file which includes the run time enforcement code out of the annotations. The generated file can then be compiled using a C compiler like gcc or clang. In addition to that the code generator can pass the generated code directly to the compiler using a pipe, which simplifies inclusion into existing build systems. The following sections describe how the run time enforcement mechanism works and how the parsing of the input file is done. A static analyzer has been implemented to help developers to identify which system calls a specific function needs. The implementation of that static analyzer is described at the end of this chapter.

4.1 System overview

Straightjacket is organized as follows:

```
/
├── analyzer
│   └── alias.txt
├── bin
│   ├── sj-analyzer
│   ├── sj-analyzer-wrapper.py
│   ├── sj-annotator.py
│   └── sj-codegen
├── codegen
├── examples
├── include
└── parser_test.sh
```

The *bin* directory holds the tools such as the code generator, the static analyzer and helper scripts. The *analyzer* directory holds the analyzer's source code and the default system call alias file. The *codegen* directory holds the code generator's source code. The *example* directory holds some annotated programs that are used as examples as well as unit tests. The *include* directory contains the header files. Finally, the root directory contains the Makefile and a script (*parser_tester.sh*) that runs the parser on all examples to verify that all input variants get parsed correctly. The *bin* directory holds the tools such as the code generator, the static analyzer and helper scripts.

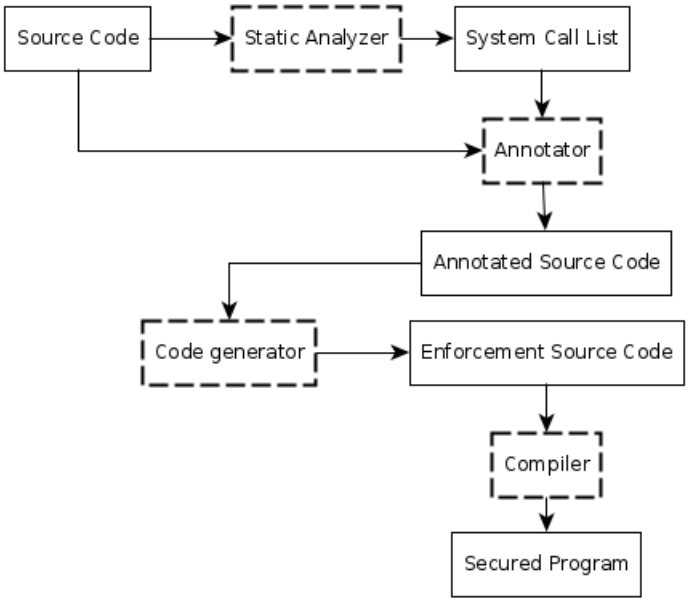

Figure 4.1: System Overview

Figure 4.1 shows the interaction between the various components. The static analyzer processes the program's source code and produces a list of identified system calls. The output and the source are then further processed by the automated annotator to produce source code with annotations. Afterwards the code generator processes the annotations and generates run time enforcement code for the annotated functions. Finally, a compiler compiles the generated code into a secured program.

Build requirements

The tools required for building Straightjacket are GNU make, bison, flex, gcc, binutils, and python. The code can be built using the "make" command and cleaned up (remove all generated files) using "make clean".

4.2 Run time Enforcement

The enforcement of the policy defined by the annotations is done using the seccomp framework, which allows a user space program to load a system call filter into the kernel to restrict system calls and their arguments for the run time of the process that set up the filter. The filter is a small program that gets executed by the kernel before running system calls. It can either be a whitelist of allowed system calls and their arguments or a blacklist of system calls and arguments that are not allowed (see chapter 2). When the process tries to call any system call not allowed by the filter the kernel will either kill the process or send a SIGSYS signal and kill the process afterwards. The return value of the filter decides whether the system call gets to run or whether the calling process should be killed by a signal.

For the whitelisting case the default action of the filter is to kill the process unless the number of the system call is in the list of allowed system calls. Blacklisting is implemented by setting the default action to allow and returning kill for system calls in the list of disallowed system calls. In order to be used to restrict access only for selected functions the functions have to run in their own processes. But given that threads on the Linux operating system are technically processes too (see chapter 2), the seccomp filter can be set for specific threads as well. [54]

The basic idea works as follows:

1. The annotated function gets renamed and replaced by a new one that has the original name.

2. The new function spawns a thread.

3. Inside the thread the seccomp filter gets set up.

4. The original function gets called.

5. The function that has spawned the thread waits for the thread to exit.

6. The return value (if any) of the original function gets returned.

A function like the one shown in Listing 4.1 gets transformed into run time enforcement code as illustrated in Listing 4.2.

Listing 4.1: Basic function

```
int my_func(int a) {
        /* Do stuff */
        return somevalue;
}
```

Listing 4.2: Transformed code

```
static int _my_func(int a);

int my_func(int a) {
        int ret = spawn_thread(_myfunc);
        wait_for_thread();
        return ret;
}

static int _my_func(int a) {
        /* Do stuff */
        return somevalue;
}
```

The serialization in the generated function hides the thread from the callers of the function. Otherwise, the introduced parallelism would require changing every caller and thus rewriting large parts of the program. The Posix thread API allows passing a pointer to a structure as argument to the thread function. [55] To deal with arbitrary function arguments and return values a structure containing the arguments of the original function gets created and passed to the thread. In addition to that a field called *retval* with the return type of the return value is added to the structure. When the thread starts the arguments from the structure are passed to the original function and its return value is stored in the *retval* field. Once the thread exits the calling function can extract the return value from the *retval* field and finally return it to the caller.

Listing 4.3: Parameter structure

```
typedef struct _my_func_params {
        int a;
        int _retval;
} _my_func_params;
```

To save memory the *retval* field gets omitted when the original function does not have any return value i.e. has the return type void. If the function has no arguments and has a no return value no structure gets generated. As an example the parameter structure for the function shown in Listing 4.1 is illustrated in Listing 4.3.

Figure 4.2 shows how the modified function behaves at run time.

The generated filters are cached to reduce the setup cost for subsequent calls of the function.

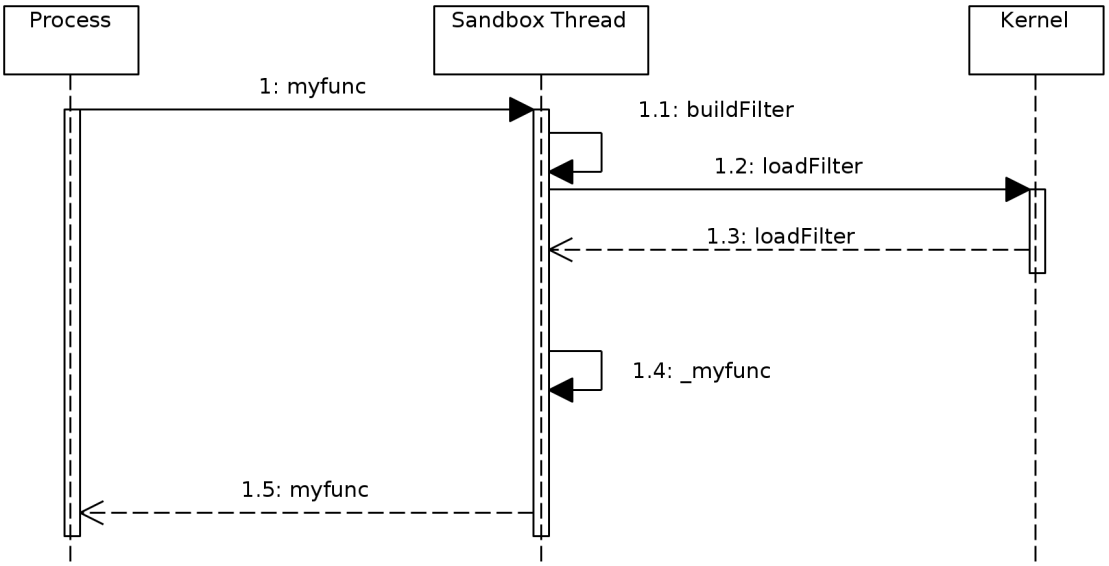

Figure 4.2: Annotated program run time sequence

4.2.1 Nested Functions

Once a filter has been set up for a thread the restrictions can only be tightened but not relaxed. That means that a function will inherit the filters set by its caller. To allow the called function to tighten the filters the system calls required to set up a filter and executing a new thread have to be always allowed, otherwise any attempt to set up a filter or start a new thread for the called function would kill the program. The generation of the filter can be done entirely in user space and therefore does not require additional system calls.

System calls are needed for loading the filter into the kernel and for locking the thread's privileges to disallow relaxing of the filter. Both can be done using the *prctl* system call, with different arguments. The latter is done by setting the *PR_SET_NO_NEW_PRIVS* flag to one. Loading the filter can be done by passing *PR_SET_SECCOMP* as a first argument and two (means user defined filter) as second argument. The third argument contains the BPF filter, in form of a pointer to the structure *sock_fprog* [1]. A detailed description can be found in the Linux kernel source file *Documentation/prctl/seccomp_filter.txt*. Newer kernel versions have a dedicated system call, called *seccomp*, for setting up the filters. This is also allowed by default to allow newer version of *libseccomp*, which use the *seccomp* system call if available, to work.

The predefined rules that allow those operations are shown in Listing 4.4

[1] man prctl

Listing 4.4: Default seccomp rules

```
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(prctl), 5,
                 SCMP_A0(SCMP_CMP_EQ, PR_SET_NO_NEW_PRIVS),
                 SCMP_A1(SCMP_CMP_EQ, 1),
                 SCMP_A2(SCMP_CMP_EQ, 0),
                 SCMP_A3(SCMP_CMP_EQ, 0),
                 SCMP_A4(SCMP_CMP_EQ, 0));

seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(prctl), 2,
                 SCMP_A0(SCMP_CMP_EQ, PR_SET_SECCOMP),
                 SCMP_A1(SCMP_CMP_EQ, 2));

seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(seccomp), 0);
```

The *seccomp_rule_add* function takes the current *libseccomp* context as the first argument, followed by the action that should be taken when the system call gets executed by the program. The third argument contains the system call that the rule applies to. The fourth argument contains the number of arguments that should be restricted. It can be set to zero in case the arguments should not be restricted at all. Finally, the arguments can be specified using the *SCMP_An* macros, where n is the index of the argument (starting at zero) that should be restricted. The first argument of the macro is the comparison mode, here only equality is used, while the second argument contains the value that the argument should be compared against [2].

Before setting up the filter a new thread has to be started to not affect the whole process. For this the POSIX thread API is used which in turn relies on the *clone* system call to create a new thread. After creating the new thread the *set_robust_list* system call gets called to set up locks that are held until after the thread exits [3]. The main thread then uses the *futex*[4] system call, which blocks until the thread releases the locks and exits. The *rt_sigprocmask*, *rt_sigaction* and *rt_sigprocmask* system calls are used to set the handling of the SIGCHILD signal which gets emitted by the kernel when the child process, in our case the thread, exits.

Because threads share the memory with their parent process the POSIX thread library changes some memory mappings for performance reasons, using the *mprotect* and *madvise* system calls. For instance it uses the *madvise* system call to tell the kernel that a particular mapped area is no longer required and thus can be paged out if the system is under memory pressure [5]. Those memory areas include the thread local storage, thread data structures and the thread stack. The former two are stored just below the stack

[2] man seccomp_rule_add
[3] man set_robust_list
[4] man futex
[5] man madvise

(or above on architectures where the stack grows upwards) to save memory allocation costs. This is explained in detail in the paper *The Native* POSIX *Thread Library for Linux* written by Ulrich Drepper and Ingo Molnar, who also implemented the current threading support in Linux. [56]

Listing 4.5: Transformed code

```
static int _my_func(int a);

int my_func(int a) {
        static int in_thread = 0;
        int ret;
        if (in_thread == 0) {
                in_thread = 1;
                ret = spawn_thread(_myfunc);
                wait_for_thread();
                in_thread = 0;
        }
        else {
                ret = _myfunc(a);
        }

        return ret;
}

static int _my_func(int a) {
        /* Do stuff */
        return somevalue;
}
```

4.2.2 Recursion

When a sandboxed function, i.e. a function that has a filter set up, recursively calls itself for instance for implementing a recursive algorithm or is otherwise re-entrant, i.e. gets called again before it returns, the system would spawn a new thread and set up the filter again for the newly called function. This is a waste of resources because it can lead to having multiple threads active at the same time even though the additional threads do not serve any purpose. This can also lead to memory exhaustion or reach the limit of maximum active processes. Both would cause the program to get terminated by the operating system. Also, generating the filter and loading it takes a small amount of time which would add up and affect run time performance negatively. To avoid that the code generator has to generate code that protects against that.

The implemented solution is to use a static variable inside the function that tracks

whether a thread is active or not. A new thread then only needs to be spawned when the flag is not set. That way it is ensured that no more than one thread is active at the same time for the same function. Based on the code from Listing 4.2 the improved code with recursion handling is illustrated in Listing 4.5.

4.2.3 Arguments

In addition to restricting the system calls that can be used restricting the arguments that can be passed to system calls can further restrict the privileges of a function. For this the arguments have to be listed in the annotation, multiple occurrences of the same system call with different arguments act as a logical *or*. That way it is for instance possible to limit the available file descriptors that can be interacted with or limit the executables that can be executed using the *execve* system call. One might also use that in blacklist mode to block execution of specific binaries that are known to be used often by attackers while the program itself does not need them. The most common examples are command shells. The syntax of the annotations is specified in section 4.3.

4.2.4 Required libraries

The run time enforcement code uses the *libseccomp* library for setting up the system call filters. It abstracts the BPF code generation and architecture specific system call numbers, resulting into easier to read and maintainable code. In addition to *libseccomp* the POSIX threading library is required for setting up threads for functions. Therefore, the generated program has to be linked against both of those libraries. The code generator does not require any external libraries at run time.

4.2.5 Debugging

When the constant *__SJ_DEBUG* is defined the generated code does not enforce the annotations by not loading the filters into the kernel. Instead, it prints the filter as pseudo code to standard error once for each called function. This can be used by developers for debugging their applications by viewing the generated filters. The constant can either be defined in the code or on the compiler command line.

4.3 Annotation

4.3.1 Syntax

The annotation consists of at least two lines that have to be added in front of a function definition. The first line has two purposes. It states that the function should be sandboxed and defines the type of the sandbox. The type can either be whitelist or blacklist. When no type is specified whitelist is assumed.

Listing 4.6: Whitelist sandbox annotation

```
#pragma sandboxed whitelist
```

Listing 4.7: Blacklist sandbox annotation

```
#pragma sandboxed blacklist
```

Listing 4.8: Default sandbox annotation

```
#pragma sandboxed
```

Listing 4.6 shows an example of an explicit whitelist annotation, while Listing 4.7 shows a blacklist annotation. The annotation in Listing 4.8 does not specify any type and therefore is equivalent with the annotation from Listing 4.6.

The lines following the sandbox type definition specify the system calls that should be white- or blacklisted. The list of system calls is a comma separated list of system call names. To improve readability the system call specification can be expressed in multiple lines. After the system call name arguments can be optionally specified in form of a comma separated list inside braces. An argument can be skipped by adding a comma without any value preceding it.

Listing 4.9: System call list annotation

```
#pragma syscalls = read, write, close
```

Listing 4.9 shows an example of a system call list specification without any arguments specified.

Listing 4.10: System call list annotation

```
#pragma syscalls = close(1), write(1,,512), read
#pragma syscalls = fstat, dup
```

Listing 4.10 shows a system call list specification with arguments. The *write* system call in that example illustrates how arguments can be skipped. It only limits the first and third argument, while skipping the second one. The example also shows that system calls with arguments and system calls without arguments can be mixed in the same annotation. The example also makes use of the possibility to use multiple system call specification lines.

An example of a complete annotation is shown in Listing 4.11.

Listing 4.11: System call list annotation

```
#pragma sandboxed
#pragma syscalls = close(1),write(1,,512),read
#pragma syscalls = nanosleep
void myfunc(int a, char *b) {
        ...
}
```

A formal specification of the syntax in form of a BNF (Backus Naur Form) grammar is stated below:

The annotation syntax

⟨*annotation*⟩ ::= ⟨*sandboxed*⟩ ⟨*newline*⟩
⟨*syscall_list*⟩

⟨*sandboxed*⟩ ::= '#pragma ' 'sandboxed ' ⟨*type*⟩

⟨*type*⟩ ::= 'whitelist' | 'blacklist' | ⟨*empty*⟩

⟨*syscall_entry*⟩ ::= '#pragma ' 'syscalls' '=' ⟨*syscalls*⟩

⟨*syscall_list*⟩ ::= ⟨*syscall_entry*⟩ | ⟨*syscall_list*⟩ ⟨*syscall_entry*⟩

⟨*syscalls*⟩ ::= ⟨*syscall*⟩ | ⟨*syscalls*⟩ ',' ⟨*syscall*⟩

⟨*syscall*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ '(' ⟨*arguments*⟩ ')'

⟨*arguments*⟩ ::= ⟨*argument*⟩ | ⟨*arguments*⟩ ',' ⟨*argument*⟩

⟨*argument*⟩ ::= ⟨*value*⟩ | ⟨*empty*⟩

The *#pragma* directive has been chosen for the annotation syntax because it is generally used for extensions that are compiler specific and should be ignored by compilers that do not support it. That way the annotations are still valid C syntax which allows editors, static analyzers and IDEs to parse the syntax without having to be modified. The annotated code can also be compiled by any compiler. Most compilers will generate a warning when encountering a pragma directive that they do not support. Those warnings can be safely ignored.

4.4 Code generator

The code generator is a parser that reads the input file and generates code based on the found annotations. It has been implemented using *flex* and *bison*.

4.4.1 Lexer

The lexer or tokenizer is generated using *flex*. A lexer is a program that reads an input stream and performs pattern matching on. For every detected pattern a token is generated. For instance such tokens can be generated for numbers, words or anything else that can be captured using a regular expression.

Flex is a tool that generates such a program from a rules file. The generated tokens can then be passed on to a parser that processes those using defined rules. Such a parser can be generated using tools like *bison* or *yacc*. The rules are defined in the *flex* rule file which includes the actions that should be performed in form of C program statements. There are two different types of tokens. There are simple tokens which are simply identifiers. For instance a rule like the one shown in Listing 4.12 generates the token *ANNOTATION* whenever a string matching the regular expression on the left hand side is found in the input.

Listing 4.12: Simple lexer rule

```
#pragma\ *?sandboxed                { return ANNOTATION; }
```

The actual matched text is thrown away i.e. the parser has no way to access it while processing the tokens. However, this is not always desirable. Suppose a parser wants to not only know that an identifier is found but needs the actual value of it. Such an identifier can be for example the name of a function argument. The information that a function argument name has been found is not sufficient for the code generator. It requires access to the name of the function argument. For such cases *flex* allows accessing the value of the matched string and provides a data structure, which is basically a C union defined in the rule file, named *yylval*, where the value can be stored. A rule that does that is shown in Listing 4.13. In addition to returning the token *ID* it stores the value of the matched text in the union. The string is copied onto the heap using the *strdup* function to make it available outside of the calling function's scope.

Listing 4.13: Lexer rule with data storage

```
[a-zA-Z_][_a-zA-Z0-9]* { yylval.val = strdup(yytext);
                         return ID; }
```

Flex does not support any parsing; it only splits the input into tokens. That means that there is no notation of non-greedy regular expressions as known from other regular expression engines like for instance perl.

Suppose the input file has the following content:

```
write(,"hello",), write(,"world",)
```

Trying to match a string between two quotes will not generate two different tokens for the strings ”hello” and ”world” but one token for the string:

```
"hello",), write(,"world"
```

This can be solved by emitting a token for the quote symbol and leaving the parsing to the parser. The parser can collect the tokens between two quote tokens and combine them into a string.

4.4.2 Parser

The parser has been generated using *bison*. The parser maintains a global structure which contains the data of the currently processed annotation. The structure consists of the name of the annotated function, the type of the function's return value, a list of the function's arguments, a list of the system calls listed in the annotation, whether the function is static or not and finally whether the annotation is a blacklist or not.

During processing the input file the parser executes the code specified in the rule file. The code fills the global data structure with the data from the current rule. When a function name is found the name of the function gets stored in the data structure. When the function is prefixed with the static keyword the *is_static* flag gets set. Likewise, when the parser finds the return value of the function the return value gets stored in the data structure. The annotation type is handled by simply comparing the type string with the string ”blacklist”. In case it matches the blacklist flag is set in the data structure otherwise it remains unset.

Function arguments require some additional processing to ensure that the type information is split from the name of the argument. For instance a function that takes a pointer as argument can have the asterisk symbol indicating that the argument is a pointer as suffix to the data type or as a prefix to the argument's name. Both are valid C syntax and have to be dealt with. This is done by concatenating the type and name strings and parsing them using a hand written function. It searches for the last occurrence of the asterisk symbol and splits the string again. The first part of the string is then stored as the data type while the second part gets stored as the name. The same has to be done for the function's return value and the function's name. But given that an annotation only applies to one function that post processing can be done right before generating the code and has not to be done in the parser. Once the type and name information is determined a structure that stores them is allocated and appended to the argument list of the annotation structure. The same applies to arrays. The information whether an argument is an array can either be appended to the type or to the name of the argument. The parser handles arrays by simply converting them to pointers and then handling them like any other pointers. This is

possible because in C an array is technically just a pointer to the first element of the array.

For system calls a separate global array and a counter gets used for tracking the arguments. The array can hold up to six argument values. The array gets initialized with NULL values. Once a system call argument is found by the parser its value is stored in the array at the position pointed at by the counter. Afterwards the counter gets incremented. When an argument has no value, which means that it should be skipped, the counter is increased without storing any value in the array. This is the reason why the counter is required. Otherwise, the number of arguments could be easily determined by finding the first NULL entry in the array. After the processing of the arguments is done the parser finds the name of the system call. At this point a new structure holding the name of the system call, the arguments and the number of arguments gets allocated and appended to the system call list in the annotation data structure. Afterwards the system call argument array and counter are reset for processing the next system call.

Once the structure is filled with the data the parser hits the main rule, which matches the whole annotation and the function declaration. At this point the *generate_code* function gets called which does the actual code generation.

The *generate_code* function simply prints the new code to standard output. At first it emits code that includes the required headers for the *libseccomp* library, *prctl* system call and for POSIX threads. The return value of the function is processed the same way as the function's arguments. That means that the type information gets split from the function name. Afterwards a structure for storing the return value and the arguments of the function, if any, gets generated. If the function takes no arguments and has no return value (type void) the structure does not get generated. The structure is then followed by a static definition of the, now renamed, annotated function. The function gets re declared as static to hide its symbol from the dynamic linker, which results into less work for the dynamic linker and avoids symbol name collisions. The purpose of the static flag in the annotation data structure is to avoid adding two static keywords to the function, which would result into a syntax error when attempting to compile the resulting code. The code generator then generates the thread function, which sets up the filter and runs the original function. The filter setup code gets generated by iterating over the system call list and printing *libseccomp* calls that enforce the defined rules. Finally, the function declaration gets printed using the new name. The detailed structure of the generated code is described in section 4.2.

Once the code for one function is generated the annotation data structure gets reset to be reused for processing the next annotation.

A parser generated by *bison* does attempt to parse the entire input file. Any content in the file that has no matching rule in the grammar triggers a syntax error. The annotations are only a small fragment of the input file ,so the remaining parts of the file

have to be dealt with. This has been solved by introducing a misc rule that captures any text before and after the annotation, that is not an annotation, and simply prints it out unmodified.

When the parser gets invoked without any arguments it reads from standard input and outputs the generated code to standard output. Another way to invoke it is to prepend it to the compiler command line. In that case it parses the compiler command line to find the name of the input file. It then reads from that file and outputs the generated code to standard output. The standard output stream gets redirected into a pipe from which a newly spawned compiler process reads. This has the advantage of easy build system integration and that no temporary files need to be created for the generated code. This mode only works under the assumption that the compiler has gcc compatible command line syntax like for instance the clang compiler. Another disadvantage of this way of operation is that when debug symbols get generated the compiler has no information about the name of the source file and therefore a debugger can no longer map symbols to positions in the source file.

4.5 System call identification

Developers do not think in terms of system calls but in functions that they call. Calling system calls is mostly completely transparent. To use a system call like *write* it is not required knowing that it is a system call, from a developer's point of view it is a function like any other one. In addition to that even for developers who know about system calls it is not always obvious that a particular function does involve one or more system calls behind the scenes. For instance the function *printf* is used to print characters to the standard output. To write to a file descriptor, like standard output, the system call *write* is required. So one might assume that in order to allow *printf* to work only the system call *write* has to be allowed. This is, at least for the *printf* implementation of the GNU C library, not sufficient. For performance reasons it uses the system call *mmap* in addition to *write*.

Another issue is dealing with complex functions that call other functions or have multiple branches that get executed based on the current program state, which might depend on external circumstances like user input. For such functions it is not easily possible for a developer to know which system calls are required during the operation of the function. Even a trial and error approach by guessing based on source code reading and testing the program is not guaranteed to be sufficient, because full test coverage is required to be sure that all possible system calls are found.

Using dynamic instrumentation with a tool like *strace*, which lists all system calls called by the running program, suffers from the same issue. To exercise all paths full test coverage is required. A static analyzer however can construct a full call graph out of the source code and therefore does not suffer from the test coverage issue. One limitation

is that not all functions are necessarily present in the source code. The function might call a function provided by a library, whose source code is not bundled with the application.

Another issue is that programs typically consist of multiple source files. A function defined in one file can call a function that is defined in another file. Analyzing source files one by one therefore would lead to limited results. Such programs however express the dependency between individual files in the Makefile. That means that a static analyzer has to follow dependencies specified in the Makefile to be able to process the code of a program, more specifically a program binary, as a unit.

4.5.1 Static analyzer

To address the needs and issues mentioned in the previous section a static analyzer has been developed.

It consists of two parts:

1. A python wrapper script that uses LLVM to generate a call graph in the dot file format.

2. An analyzer that parses the generated call graph and identifies system calls.

The dot file format is a simple text file that defines nodes in a graph and their connections to each other. It is mainly used for generating graphical representations of the graph in various formats. The simplicity of the format allows easy parsing which makes it suitable for other purposes as well.

Python wrapper script

The python wrapper script gets injected into the build process and uses the clang compiler to generate LLVM IR (intermediate representation) code whenever a C file gets compiled. LLVM IR is a similar concept as assembly language. It is intermediate code generated by the compiler before finally translating it into assembler and finally machine code. In addition to that it generates LLVM bit code whenever a linker command gets called. Generating LLVM bitcode out of LLVM IR files is similar to linking multiple object files together to generate a program binary. Finally, the opt tool is used to generate the dot call graph for further processing by the analyzer. This process is illustrated in Figure 4.3.

Suppose a program that gets compiled using the commands shown in Listing 4.14.

Listing 4.14: Sample build process

```
CC=gcc
$(CC) -O2 -c program.c
$(CC) program.o -o program
```

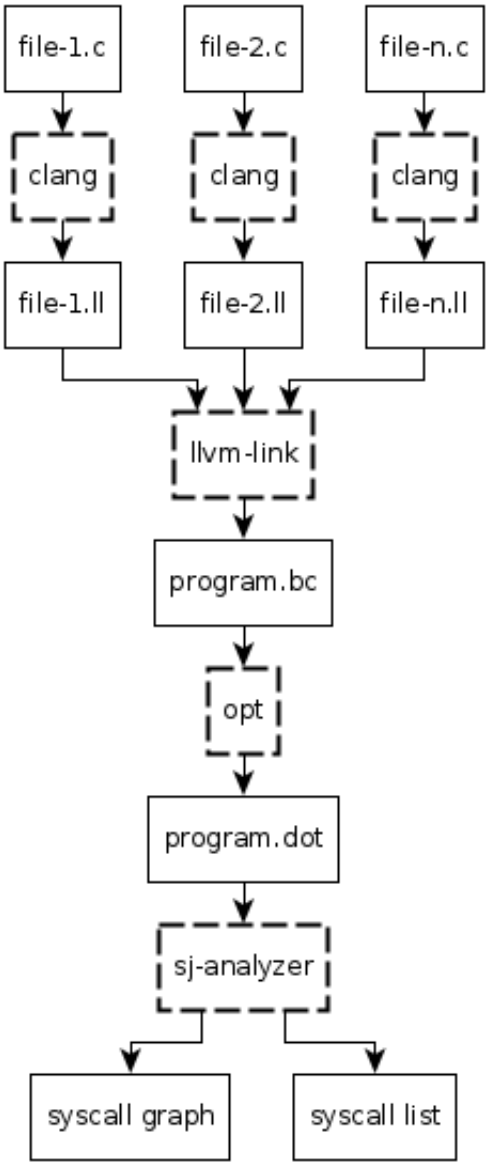

Figure 4.3: Static analyzer build process injection

The first command compiles the input file *program.c* into an object file named *program.o*, while the second command takes that object file as input and generates an executeable called *program*. The python script hijacks this process and injects three additional commands. Those commands are shown in Listing 4.15.

Listing 4.15: Injected LLVM commands

```
clang -S -emit-llvm -O2 -c program.c
llvm-link program.ll -o program.bc
opt -analyze -dot-callgraph < program.bc
```

The first command compiles the input file *program.c* into a LLVM IR file called *program.ll*. The second command takes the generated *program.ll* as input and generates a LLVM bit code file called *program.bc*. Finally, the third command passes the LLVM bit code to the opt tool which generates a call graph file out of it named *callgraph.dot*. Because a project can contain multiple binaries that get build from the same source code tree, the python script renames the *callgraph.dot* file to *program.dot*. This ensures that the file name is unique and easy to relate to a specific binary.

The wrapper script does not replace the actual build commands it only supplements them. This is done because otherwise the build process might halt with an error due to a missing dependency like for instance an object file that should have been generated but got replaced with the command that creates the LLVM IR file.

Analyzer

The analyzer parses the dot file using a parser generated by *bison* and *flex*. It maps the graph to a hash table, where the name of a function is used as the key and a linked list of other nodes, the called functions, packed into a C structure, as value. A node in the graph also has a flag that gets used for recursively collecting system calls for a sub tree. This data structure allows efficient look up of the list of called functions for a given function.

In addition to the nodes a list of edges gets stored into a separate hash table. An edge is a data structure that holds a path, which is a string consisting of the id of the starting and ending vertices, and a visited flag.

The *uthash* library is used as hash table implementation. The *uthash* library has the advantage that it has no run time requirements. It only provides a set of preprocessor macros and data structures.

The analyzer uses the generated hash table to find the node of the function to be analyzed. It then traverses the graph using depth first search and marks all edges it visits as visited. This is required to avoid infinite loops when a recursive function call is found. The edges are used for the visited tracking to make sure that every path gets

visited, marking only the nodes as visited is not sufficient because a function might get called multiple times from different branches.

The analyzer supports two output modes which can be chosen by the user by passing a command line argument. The first mode displays the call graph as a graph in text form. In that mode every function is printed out with an indentation level that matches the depth of its occurrence. If a function is a system call it gets highlighted in the output. The second mode displays a list of system calls for each function found in the program. That list is generated by a separate function that gets called for each node. The function takes the id of the node and marks all visited nodes with the given id. This is required to avoid infinite loops. The system calls found in the sub graph are then added to a hash table to ensure uniqueness. Once the whole graph is processed the keys from that hash table get printed out for every node.

To know whether a function is a system call or not the analyzer needs a list of system calls. Because system calls are defined and implemented in the kernel the system call definitions have been taken from the kernel's source code. The Linux kernel source files *arch/x86/syscalls/syscall_32.tbl* and *arch/x86/syscalls/syscall_64.tbl* contain, among other data, a list of all system call names present in the kernel version it has been copied from. Those two files are merged into a *syscall.tbl* file which contains the data from both. To avoid having to parse the *syscall.tbl* file each time at run time the file gets parsed using a python script at compile time. That script generates a *is_syscall* function that the analyzer can use at run time to detect whether a specific function is a system call. The *is_syscall* function contains an array of all system call names that gets stored into a static hash table at the first time the function gets called. This avoids having to iterate over the whole system call list each time the function is called. The run time of the function is therefore constant instead of linear in the number of system calls.

The static analyzer allows mapping functions to specific system calls to handle library functions. For instance one can define that the (library) function *foo* is an alias for the system calls *open*, *read* and *close*. Those aliases can be specified in a text file that the analyzer reads. The information from the file is then used to identify more system calls.

The file format consists of multiple lines where each alias is defined in one line. An alias is specified as function name followed by a colon and a comma separated list of system call names. Lines starting with a hash are considered comments and therefore ignored by the parser. Those can be used to structure the file. For instance by grouping the functions by library and adding a comment to each block to note which library the functions belongs too. Empty lines and whitespace gets ignored. This file is read and parsed at run time to allow easy modification without having to recompile the analyzer. Because of the simplicity of the file format *bison* and *flex* are not involved in the parsing. Instead the parsing is done by a simple *getline* and *strtok* loop.

Listing 4.16 shows an example of such an alias file.

Listing 4.16: Example aliases

```
# stdio
printf:mmap,fstat,write,map2,fstat64
puts:mmap,fstat,write,map2,fstat64
fflush:write
perror:dup,fcntl,fstat,mmap,write,close,map2,fstat64
fgets:fstat,mmap,read,map2,fstat64

# misc
sleep:nanosleep
```

Limitations

As noted in section 4.5.1 calls to external libraries are not detected directly by the analyzer and are handled by the alias mechanism. Another limitation is that by being a static analyzer it cannot trace dynamically generated code. The same applies to code that assigns function pointers at run time. The detection of system calls is limited in those scenarios.

4.6 Automated annotation

When trying to annotate an existing program the static analyzer can be used to identify system calls, but the actual annotation has still to be done by the developer based on the results. Under the assumption that the static analyzer has been able to identify all system calls used by an application, the code can be annotated automatically. While in practice the goal should be to annotate only functions that process external input or are likely to be vulnerable for other reasons, the automated annotator script developed for this thesis attempts to annotate every function that uses system calls. While the script has been developed for the evaluation of the solution, which is done in chapter 5, it can still be useful in other situations. For instance for generating initial annotations and then either extending or trimming them down manually.

The annotator is written in python and works by processing the list output of the static analyzer. It can be either pointed at a specific C file or a directory that contains a program, consisting of multiple C files. The script builds a list of C files it has to process, in case of single file the list contains only one entry, otherwise it recursively searches for C files in the target directory. Once the list of files has been generated the static analyzer output needs to be parsed.

In the output of the static analyzer each function that uses system calls is listed followed

by a comma separated list of system call names inside square brackets. Listing 4.17 shows such an output line.

Listing 4.17: Example analyzer list output

```
main[open,write,close]
```

The parsing is done using a regular expression. The result is then stored inside a dictionary (hash table in python) using the function name as a key and the list of system calls as value.

To avoid having to parse C files manually the *gccxml* tool has being used in the first implementation. It is a gcc extension that uses gcc's C++ parser to parse the input file and outputs the abstract syntax tree as an XML file. To avoid unnecessary disk access the *gccxml* output is read directly via a pipe. The generated XML file contains among others function declarations and their positions in the files. A C file is assigned a unique id. The python script uses the *minidom* library to find all *File* tags inside the XML file and extracts the id of the currently processed file from the *id* attribute. Once the id of the file has been determined it goes on to find all *Function* tags inside the XML that have the id of the current file assigned to them and are present in the dictionary built from the static analyzer output. A new dictionary using the function name as key and the line number where it is declared as value is then constructed.

After the parsing is done the original C file is read line by line and stored into a buffer. When a function declaration that should be annotated is found the annotation is prepended to it in the buffer. Finally, the buffer is written to a temporary file. The temporary file is then renamed to overwrite the original file. The "use temporary file and rename" pattern is used to make sure the operation is atomic. In case of an error before the process is finished the original file will stay present.

The approach based on *gccxml* turned out not work in some circumstances because *gccxml* uses gcc's C++ parser internally. While C and C++ are similar languages there are some constructs that are valid in C but not in C++. To overcome this limitation *gccxml* has been replaced with a custom gcc plugin that outputs the position and file data as a JSON string. The plugin is written in python using gcc's python bindings and uses gcc's C parser internally. It registers a callback for the *PLUGIN_PRE_GENERICIZE* event to capture function declaration locations. Other than exchanging the call of *gccxml* with the custom gcc plugin and replacing the XML parsing with JSON deserialization the automated annotator had not been modified.

To improve the speed of processing the annotator uses a thread pool to run multiple processing routines simultaneously, when the input is a directory that contains multiple C files. To not overload the system the number of concurrent threads is limited to the number of processor available on the system.

The resulting program then has every function that uses system calls, which could be identified by the static analyzer, annotated with the system calls that it uses. The developer can then do further fine tuning manually.

4.7 Usage

This section describes how STRAIGHTJACKET can be used by a developer. The interaction between the tools is explained in section 4.1. The tools have been developed in a way such that their usage is at least invasive as possible i.e. can be integrated in existing developer workflows with little effort.

Code generator

The code generator can operate in two modes. It either reads from standard input and outputs the generated code to standard output as shown in Listing 4.18, or it acts as a wrapper for a compiler command as shown in Listing 4.19. The latter passes the generated code to the compiler via a pipe which continues the compilation process. When used on a file without annotations the code generator simply outputs the input file unmodified or passes it on to the compiler. It therefore can be simple used for all files regardless of whether they contain annotations or not.

Listing 4.18: File transformation example

```
sj-codegen < inputfile.c > outputfile.c
```

Listing 4.19: Compiler wrapper invocation example

```
sj-codegen gcc inputfile.c -o program -lseccomp -lpthread
```

The wrapper mode can be used for simple integration with existing Makefiles. The compiler command just has to be prepended with the code generator command. In addition to that the libraries *libseccomp* and pthreads have to be added to the linker flags. The annotated code can also be compiled with any standard compiler so it's easy to switch between a sandboxed and non-sandboxed program, for debugging or comparison purposes, without having to maintain two copies of the code.

Static analyzer

The static analyzer runs on call graphs in the dot file format generated by LLVM. Those can be generated by a wrapper script that is supplied with the static analyzer. The analyzing is therefore a two-step process. At first the script has to be prepended to the compiler command line. It generates a dot call graph file for each binary. Those files can then be supplied to the static analyzer to identify system calls used by the binaries. Since the analyzer is not supposed to be part of the build system, modifying the Makefiles is not strictly necessary. It can be using by simply setting the *CC* (environment) variable

to ”sj-analyzer-wrapper.py gcc” where gcc is the compiler that should be used as show in Listing 4.20.

Listing 4.20: Analyzer wrapper invocation example

```
make CC="sj-analyzer-wrapper.py gcc"
```

The analyzer itself can then be used on the generated dot files that are placed next to the compiled binaries in the build folder.

Listing 4.21: Analyzer invocation syntax

```
sj-analyzer [-a aliasfile.txt] [-l] [-g] [functionname]
```

Listing 4.22 shows the invocation syntax of the analyzer. The option ”-a” can be used to supply a custom system call alias file. The ”-l” option outputs the results as a list, while the ”-g” option outputs the results as a graph. The last argument is the name of the function that should be analyzed, if none is passed main is used. All arguments are optional the default output mode is the graph display. The input is read from standard input.

The list output can be further passed to the automated annotator described in section 4.6.

Listing 4.22: Automated annotator invocation

```
sj-annotator.py target analyzerlist.txt
```

The automated annotator takes the target filename or directory as first argument and a file containing the list output from the analyzer as second argument.

CHAPTER 5

Evaluation

In this chapter the developed solution is evaluated in terms of effective exploit mitigation and performance overhead. The former is done by analyzing a set of publicly documented vulnerabilities. Limitations of the approach are also discussed in this chapter. The performance evaluation is done using the *SPEC CPU2006* benchmark suite to simulate some real world usage. The results are then compared with an unmodified copy of the *SPEC CPU2006* suite.

5.1 Exploit mitigation

In this section the effectiveness of the exploit mitigation is evaluated on different types of exploits that allow arbitrary code execution by an attacker. The exploits have been selected in a way to cover multiple scenarios like local and remote vulnerabilities. All of the analyzed vulnerabilities have CVE entries assigned. Because the tools presented in this thesis work on C code written for the Linux operating system, only exploits where the vulnerable software is still available, open source, runs on the Linux operating system and written in the C programming language were considered. Each section describes a vulnerability followed by a system call identification of the system calls required by the vulnerable function. Afterwards the system calls are analyzed whether they allow running arbitrary executables, connecting back to an attacker, binding to a network port or do any other kind of malicious actions like coping files to locations accessible by the attacker.

5.1.1 mcrypt - CVE-2012-4409

The tool *mcrypt* is a command line encryption tool. A vulnerability in version 2.6.8 and earlier is documented as *CVE-2012-4409*. An attacker can execute arbitrary code by sending the user a malicious file to decrypt. The function *check_file_head* does not validate the length of the salt before reading it from the input file into a static buffer. It

only verifies that the length of the salt is greater than zero but does not check whether it fits into the target buffer. The consequence is that a long salt can overflow the buffer that is used for the header parsing and therefore run a stack based buffer overflow exploit. The function itself is relatively simple it only requires file I/O in form reading from a file stream and writing error messages to standard error if it encounters any errors.

Listing 5.1: Annotated vulnerable function - check_file_head

```
#pragma sandboxed
#pragma syscalls = fstat64,mmap2,fstat,mmap,read,write(2,,)
int check_file_head(FILE *fstream, char *algorithm, char *mode,
                    char *keymode, int *keysize, void *salt,
                    int *salt_size)
```

The annotation in Listing 5.1 uses a white list approach to limit the function to just the system calls required for operation. In addition to that the first argument of the *write* system call is limited to the file descriptor two, which is standard error.

With the available set of system calls the shell code in the payload of the malicious file cannot run arbitrary commands using the *execve* system call. Likewise, any network connection using the *connect* system call or even binding a socket using the *bind* system call is not possible. The attacker could for instance read files from the user's home directory (or any other user readable file) and send the contents to some place over the network, but the lack of networking system calls prevents that. A DoS (denial of service) attack to fill up the user's hard drive is also not possible unless the user redirects standard error to a file.

Due to of the simplicity of the vulnerable function the system calls that can run in the context of the function can be restricted in a way to severely limit the shell code that can run in its context. A mcrypt version annotated with the annotations from Listing 5.1 would have prevented most exploitation attempts.

5.1.2 gif2png - CVE-2009-5018

The image format conversation tool *gif2png* did not validate the length of command line arguments prior to version 2.5.3, which leads to a vulnerability that can be exploited by passing long file names. While being a local exploit it can be potentially exploited remotely because some web services use it in CGI scripts to convert images. A malicious file name can be used to run arbitrary code with the privileges of the application. The vulnerability is due to using the unsafe *strcpy* function inside the main function, instead of *strncpy* which allows passing in the size of the target buffer. An annotation for the main function that uses a white list to limit the system calls to those required for operation is shown in Listing 5.2.

Because the vulnerability is inside the main function the available system calls are effectively all that are required by the application. Opening network connections or executing commands are blocked by the annotations. However, all system calls required

for file operations are available to the attacker. The attacker can therefore overwrite or even delete files that the user running the *gif2png* tool has access to. File reading is possible but without a way to open a network connection it is not possible to transfer the result directly to the attacker, unless there is a place where the attacker and the program have access to, like for instance a directory accessible by a web server or an attacker owned directory on a multi user system.

This vulnerability shows a case (vulnerability inside the main function) where STRAIGHTJACKET does not provide any additional security compared to traditional sandboxes that confine the whole application.

Listing 5.2: Annotated vulnerable function - main

```
#pragma sandboxed
#pragma syscalls = mmap, fstat, write, mmap2, fstat64, exit, ioctl
#pragma syscalls = read, open, close, mremap, dup, fcntl, unlink
int main(int argc, char *argv[])
{
...
```

5.1.3 dproxy - CVE-2007-1465

dproxy is a DNS caching server used by some routers and wireless access points. A vulnerability in version 0.1 up to 0.5 is documented as *CVE-2007-1465*. Inside the main function a UPD packet is copied using *strcpy* into a local buffer called *query_string* which has a fixed size. That means that a large malicious UDP packet can be used to overflow the buffer and execute arbitrary code.

Listing 5.3: Annotated vulnerable function - main

```
#pragma sandboxed
#pragma syscalls = open, dup, fcntl, fstat, mmap, write, close
#pragma syscalls = mmap2, fstat64, read, ioctl, exit, socket
#pragma syscalls = getpid, syslog, bind, fork, signal, recvfrom
#pragma syscalls = flock, rename, time, unlink, lseek, sendto
int main(int argc, char **argv)
{
...
```

Because the vulnerability is inside the main function the available system calls are effectively all that are required by the application. An annotated main function with the required system calls is shown in Listing 5.3. The set of system calls available allow the injected shell code to bind to network ports, and/or rename, delete or overwrite accessible files.

In addition to that reading files and sending the results over the network to an attacker is possible with those annotations due to the availability of the *open*, *read*, *bind*, *write*, *sendto* and *socket* system calls.

This vulnerability is similar to the one described in subsection 5.1.2, but given that the set of system calls is larger it allows more attack scenarios.

Listing 5.4: Annotated vulnerable function - prepare_reply

```
#pragma sandboxed
#pragma syscalls = write,open,close,fcntl,stat,fstat
int prepare_reply(struct request *r)
{
...
```

5.1.4 mathopd - CVE-2003-1228

mathopd is a small single process web server for UNIX operating systems that also supports executing CGI scripts. A vulnerability in version 1.2 up to 1.5b13 has been documented as *CVE-2003-1228*. The function *prepare_reply* copies data into a local buffer called *buf* using *sprintf*, which does not perform bound checks. The buffer has a fixed length therefore a malicious request can overflow the buffer and run arbitrary code.

An annotation for the vulnerable function that uses a white list to limit the system calls to those required for operation is shown in Listing 5.4. The set of available system calls is limited. A shell code that tries to execute arbitrary commands doesn't work with that set of system calls. Also connect back shells, like the one used in the sample exploit from exploit-db[1], do not work because the *connect* system call is not available. Likewise, a shell code that binds to a port won't work either due to the lack of the *bind* and *socket* system calls. The system calls *open*, *write* and *close* are available which means that the injected shell code can overwrite arbitrary files that the web server has access to. That can be used for instance to overwrite a web page served by the web server. Another possible attack vector would be to write out a CGI script which does malicious actions and execute it afterwards. However, a CGI script needs to be executeable in order to be executed by the web server. The shell code would need one of *chmod*, *fchmod* or *fchmodat* system calls to do that, but neither is allowed.

The limited set of system calls available to the attacker would have prevented most exploitation attempts. Despite the application being a web server that requires network access limiting the access to specific functions blocks networking based exploits. A traditional sandbox that confines the whole application would have to grant access to the network system calls to allow the application to perform its tasks.

[1] www.exploit-db.com

5.1.5 MiniUPnPd - CVE-2013-0230

MiniUPnPd is an UPnP (Universal Plug and Play) daemon for UNIX operation systems, which is being used on many devices. A vulnerability in version 1.0 in the handling of SOAP messages is documented as *CVE-2013-0230*. The vulnerability is inside the *ExecuteSoapAction* function which copies a user supplied message into a local buffer using the *memcpy* function. While the size of the buffer is fixed to 2048 bytes, the number of bytes copied into the buffer can be controlled by an attacker because it is derived from the message content. Therefore, a malicious SOAP request can be used to execute arbitrary code by exploiting the buffer overflow vulnerability. The vulnerability has been analyzed in detail by Dejan Lukan. [57]

An annotation for the vulnerable function that uses a white list to limit the system calls to those required for operation is shown in Listing 5.5. As the annotation shows the vulnerable function only requires two system calls for operation. Those two system calls, *syslog* and *send*, can only be used to log data into the system log or to send data over the network to an already opened socket. A possible attack scenario would be to try to send arbitrary data from the server's address space back to the attacker using the *send* system call. This could be used to extract sensitive information out of the process if there are any. Traditional shell codes that try to open a shell, bind to a port or connect to an arbitrary host would fail. File system operations are also not possible with the set of available system calls.

This is a similar case to the *mathopd* vulnerability described in subsection 5.1.4, which shows the advantage of Straightjacket compared to traditional sandboxes, namely that the set of available system calls in the attack context is significantly reduced, causing most exploitation attempts to fail.

Listing 5.5: Annotated vulnerable function - ExecuteSoapAction

```
#pragma sandboxed
#pragma syscalls = syslog,send
void
ExecuteSoapAction(struct upnphttp *h, const char *action, int n)
{
...
```

5.1.6 tinc - CVE-2013-1428

tinc is a VPN (virtual private network) program that supports compression and encryption. *CVE-2013-1428* documents a vulnerability in the versions prior to 1.0.21 and between and 1.1 prior to pre7. The vulnerability is inside the *receive_tcppacket* function which copies input data into a local buffer, located inside the structure *vpn_packet_t*, using the *memcpy* function without checking whether the data fits into the local buffer.

The resulting buffer overflow can be used by an attacker to execute arbitrary code in context of the application. While the vulnerable function only consists of a few lines the call graph from the function on requires many system calls for operation. The *system* function is used to execute shell commands to set up routing tables using the route command. The *system* [2] function does that by forking a new process and passing the passed in command to a shell. That means that in addition to the system calls required for operation of the function itself all system calls required by the shell and the commands it runs would have to be included in the annotation. In addition to that the function requires system calls like *read*, *write*, *open* and *connect* which means that file system access and connecting back to an attacker is possible.

The consequence is that there is no practicable annotation to restrict the set of available system calls and at the same time block common exploitation scenarios. That shows a limitation of STRAIGHTJACKET namely that function level restrictions do not help if individual functions require a broad set of privileges.

5.1.7 lcms - CVE-2013-4276

lcms is an open source color management engine which consists of multiple tools for handling color formats and working with icc profiles. *CVE-2013-4276* documents two vulnerabilities that can be exploited by a malicious icc file or tiff image. The first vulnerability affects the *icctrans* tool. It is inside the functions *PrintRange* and *GetLine*. The former uses the *sprintf* function without validating the length of the input causing a buffer overflow. The latter uses the *scanf* function to read data into a buffer which does not perform any bound checks on the input.

Listing 5.6: Annotated vulnerable function - PrintRange

```
#pragma sandboxed
#pragma syscalls = mmap, fstat, write(1,),mmap2, fstat64
void PrintRange(const char* C, double v, double Range)
{
...
```

Listing 5.7: Annotated vulnerable function - GetLine

```
#pragma sandboxed
#pragma syscalls = mmap, fstat, read(0,),mmap2, fstat64
void GetLine(char* Buffer)
{
...
```

The annotations in Listing 5.6 and Listing 5.7 show the set of system calls required for the operation of both functions. The only difference between the two sets is that the

[2] man system

PrintRange function can write messages to standard output while the *GetLine* can read messages from standard input. Neither can perform any network operations. Accessing already open files is possible using the *mmap* system call by guessing an already open file descriptor. The guessing can be done using a simple loop that tries all possible file descriptors. This is possible because file descriptors are simply integers. However, there aren't any interesting file descriptors that can be accessed other than the input file itself which is already provided by the attacker.

The second vulnerability affects the *tiffdiff* tool. It is inside the *CreateCGATS* function which builds a string buffer out of the two arguments and stores the result into a local variable. It does that using the *sprintf* function which does not perform any bound checks, which results into a buffer overflow.

Listing 5.8: Annotated vulnerable function - CreateCGATS

```
#pragma sandboxed
#pragma syscalls = open,read,write,close,mmap
#pragma syscalls = fstat,time,mmap,fstat64
static
void CreateCGATS(const char* TiffName1, const char* TiffName2)
...
```

The annotation in Listing 5.8 shows the required system calls required inside the vulnerable function. The system calls can be used in the same way as the ones available for the first vulnerability with the exception that the *read* and *write* system calls are not restricted to specific file descriptors. That means the attacker can overwrite arbitrary files accessible by the attacked user or copy files to locations accessible by him like for instance a directory served by a web server or a shared directory accessible on a multi user system.

Listing 5.9: Annotated vulnerable function - logurl

```
#pragma sandboxed
#pragma syscalls = syslog
void logurl(struct clientparam * param, char * req, int ftp)
{
...
```

5.1.8 3proxy - CVE-2007-2031

3proxy is a HTTP proxy server. *CVE-2007-2031* documents a vulnerability in the versions 0.5 to 0.5.3g, and 0.6b-devel that allows a remote attacker to execute arbitrary code. The bug is a buffer overflow in the *logurl* function which copies the request into a fixed size buffer using *strcpy*. Because *strcpy* does not perform any bound checks, the request can overflow the buffer.

As the annotation in Listing 5.9 shows, the only system call required by the function is *syslog*. That means that shell code injected in the context of this function can only write data to the system log. If the attacker has access to the system log he could try to write arbitrary content from the process's address space into the system log to extract sensitive information. A remote attacker is unlikely to have such access though.

This is a vulnerability in a relatively simple function and therefore most exploitation attempts would have been prevented by Straightjacket.

5.1.9 Generic attacks

The previous sections discussed possibles to exploit the specific applications. This section handles generic ways to attack the underlying sandbox technology.

Attacking the kernel

To circumvent the kernel's enforcement of seccomp one has to attack the kernel itself. The kernel exposes over 300 system calls each of them which could be used to exploit a vulnerability. However, the set of system calls available to the vulnerable applications is a only a small fraction of that. Therefore, the likelihood of a successful kernel attack is significantly reduced. [8] [7]

Shared address space

Because threads share the address space with the parent process (and other threads) one could try to take advantage of that to run a successfully attack. A global attack target is the global offset table (GOT). It is a table that holds the addresses of library function calls. A vulnerability that allows an attacker write to arbitrary memory addresses can be used to overwrite such an entry and point it to an arbitrary function. [58] To successfully exploit this, the attacker has to find a library function that runs in a context where the system calls required for the attack are allowed. This can be made harder by enabling full RELocation Read Only (RELRO) when compiling the binary. In general existing exploit mitigation techniques like stack canaries or address space layout randomization (ASLR) and Straightjacket are not mutually exclusive. [59] They can and should be used together to further reduce the likelihood of a successful attack.

5.1.10 Conclusion

Table 5.1 summarizes the results. The first column shows the CVE number of the analyzed vulnerability. *CVE-2013-4276* (lcms) is listed twice because it actually covers two vulnerabilities. The subsequent columns show whether a particular action like executing commands, binding to a network port or connecting back to an attacker's machine would have been possible in the context of the vulnerable functions by the injected shell code. The last column shows whether file system access is available in the context of the vulnerable functions.

Vulnerability	Execute commands	Network bind	Network connect	File system access
CVE-2012-4409	No	No	No	Limited
CVE-2009-5018	No	No	No	Yes
CVE-2007-1465	No	Yes	No	Yes
CVE-2003-1228	No	No	No	Yes
CVE-2013-0230	No	No	No	No
CVE-2013-1428	Yes	Yes	Yes	Yes
CVE-2013-4276 (1)	No	No	No	Limited
CVE-2013-4276 (2)	No	No	No	Yes
CVE-2007-2031	No	No	No	No

Table 5.1: Vulnerability analyzes summary

The results show that STRAIGHTJACKET would have prevented exploitation attempts in most cases. It also highlights some limitations of the approach. Preventing an exploit here means not allowing the attacker to gain control of the system. The exploits still cause denial of service by causing the vulnerable programs to terminate.

By restricting access on function level STRAIGHTJACKET does not provide any additional protection compared to a traditional sandbox for vulnerabilities inside the main function. This has turned out during the analysis of the vulnerabilities *CVE-2009-5018* (gif2png) and *CVE-2007-1465* (dproxy). Nevertheless, exploits would have been prevented for both cases. The worst case was the vulnerability *CVE-2013-1428* (tinc), where the vulnerable function required a large set of system calls to the extent where a practical annotation was not possible. As a consequence STRAIGHTJACKET wouldn't have prevented exploitation attempt for that vulnerability. This also was the only case where executing shell commands were permitted.

On the other hand other vulnerabilities like *CVE-2003-1228* (mathopd) and *CVE-2013-0230* (MiniUPnPd) demonstrated the strength of the approach. Network based system calls are not permitted in the vulnerable function's context despite the applications requiring network access for general operation.

File system access was the most common attack vector that was still available to attackers. This is the case because the *read* and *write* system calls are commonly used and therefore required by most functions. In two cases that access could be limited by restricting the file descriptor arguments of the *read* and *write* system calls. This however only works for static file descriptors like standard output, standard error and standard input.

The likelihood of kernel level attacks is reduced by the limited set of available sys-

tem calls. Other exploit mitigation techniques such as stack canaries, full RELRO and ASLR should be enabled in conjunction with STRAIGHTJACKET to further enhance the application's security.

5.2 Performance

5.2.1 Benchmark selection

The performance has been evaluated using the *SPEC CPU2006* benchmark suite. Because STRAIGHTJACKET acts on C source files; the benchmarks not written in C have been excluded. In addition to that the benchmark *400.perlbench* has been excluded as well because limiting the system calls for an interpreter that runs arbitrary code is in most cases not feasible, due to limitations of the static analyzer and how those kind of programs work. An interpreter that uses a just in time compiler (*JIT*) generates code at run time and executes it afterwards. The static analyzer cannot know beforehand which system calls that code is going to use. The perl interpreter does not use a *JIT*. However, it compiles programs into a tree of operation (OP) structures that contain function pointers that get called at run time. This is explained in detail in a comment at the top of the *op.c* file from perl's source code. The assignment of those pointers happens at run time while parsing the user supplied program and therefore cannot be determined statically by the static analyzer. In addition to that it makes use of the *setjmp*[3] and *longjmp*[4] functions to implement exception handling. Those allow arbitrary jumps from a function to a parent function in the call graph. The *LLVM* based static analyzer does not add an edge in the call graph for this kind of control flow transfer. Without being able to determine the control flow the static analyzer cannot accurately determine the system calls required by individual functions.

The used benchmarks therefore are:

- 401.bzip2
- 403.gcc
- 429.mcf
- 445.gobmk*
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref

[3] man setjmp
[4] man longjmp

- 433.milc
- 470.lbm
- 482.sphinx3

Each benchmark, expect *445.gobmk*, has been run three times in four different configurations. Because the *445.gobmk* decides which functions it calls dynamically using function pointers, it's not possible to detect all system calls for every function using the static analyzer (section 4.5.1). As a consequence the *445.gobmk* benchmark has only been run in with the first two configurations. The reported results are based on the median values of those three runs. The computation has been done by the *SPEC CPU2006* benchmark suite. It reports median values in addition to the raw data when multiple iterations are run. The results show the overhead in percent compared to the base run. To improve readability of the diagrams the number prefix of the benchmark names have been omitted from the labels. The benchmark configurations are listed below.

Unmodified
: The benchmark has run without any modifications and acts as baseline.

Only main
: Annotations were only applied the main function. This works like a traditional sandbox but without any need for external programs at run time and therefore is an interesting use case. In addition to that it measures the overhead of the BPF filter.

User input annotation
: Annotations were applied to all functions that process user input in addition to the main function. Functions that process user input are the primary attack target and therefore are the ones that are expected to be annotated in real world usage. "Function that processes user input" has been approximated by a heuristic. The heuristic simply picks functions that use the *read* system call because this is the only source of user input that the *SPEC CPU2006* benchmark suite uses.

Full annotation
: Annotations were applied to all functions that use system calls. This is not a realistic case but it shows the performance overhead of overusing the annotations.

Test system

The tests have been run on a FUJITSU LIFEBOOK U904 laptop equipped with 10GB of RAM and a INTEL Core i7-4600U processor running the Linux distribution Fedora 21 x86_64 with the kernel 3.18.9. The benchmarks have been compiled using gcc 4.9.2 with the default flags for the x86_64 processor architecture.

5.2.2 Setup and configuration

The configuration file has been based on the *Example-linux64-amd64-gcc43+.cfg* file provided by the *SPEC CPU2006* benchmark suite. By default, *SPEC CPU2006* refuses to run benchmarks whose source files have been modified. However, STRAIGHTJACKET operates on source code so this check had to be disabled in the configuration file by setting *strict_rundir_verify* to zero. To use the STRAIGHTJACKET code generator when compiling the benchmarks the variable *CC* has been pointed to a script that wraps the code generator. The script is shown in Listing 5.10. The script just prefixes the command line arguments with "gcc " before passing them to the code generator.

Listing 5.10: SPEC compiler wrapper script

```
#!/bin/sh
sj-codegen gcc $@
```

Because the run time code requires both the *libseccomp* and pthread libraries the variable *EXTRA_LDFLAGS* in the file *benchspec/Makefile.defaults* has been set to "-lpthread -lseccomp".

5.2.3 Results

Only main

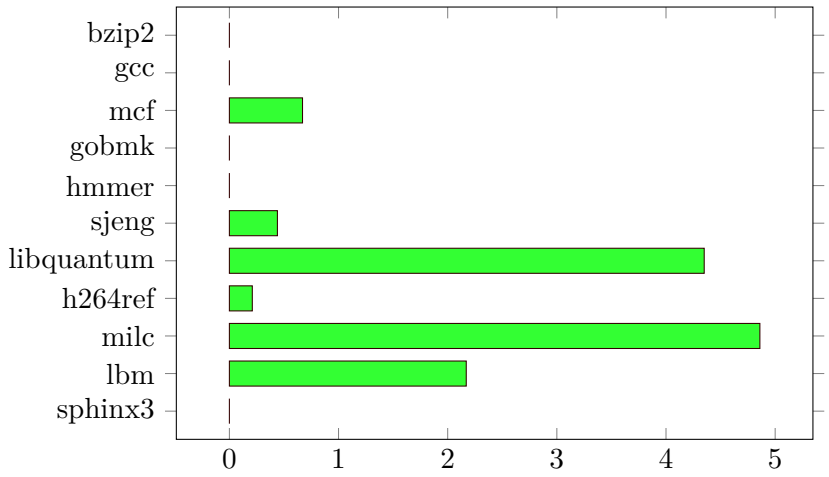

Figure 5.1: Performance overhead in % - Only main

Figure 5.1 shows results of the benchmark runs with only the main function annotated. The overhead ranges from non-measurable to slightly below 5%. An overhead of 5% as seen in the *433.milc* benchmark seems rather high for just annotating the main function, but as further results show this overhead seems to not grow further when annotating more functions. The *433.milc* benchmark does a lot of memory allocations using the *malloc* and *calloc* functions. Both functions require system calls and therefore suffer from slight

performance degradation from the BPF filter execution. The overhead sums up during the lifetime of the benchmark. Most other program's performance is only influenced by the the thread creation and filter generation during start up. For long running programs this influence is negligible. The median value is as low as *0.21%* which means in general one can expect no noticeable overhead when annotating the main function.

Full annotation

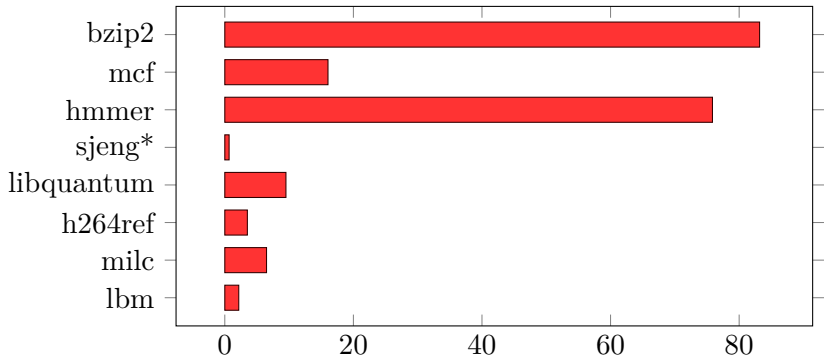

Figure 5.2: Performance overhead in % - Full annotation

The result of the benchmark runs with all functions that use system calls annotated functions is shown in Figure 5.2. Problems where encountered with three benchmarks.

The first problem occurred when trying to run the *403.gcc* benchmark with all functions that use system calls annotated. Almost every function in gcc's source tree can optionally print out debug information. This requires system calls and therefore the static analyzer marked all of them for annotation. The end result was that almost every function ended up spawning a thread to do its work. As a consequence the benchmark seemed to run forever and the run had to be canceled. A test run with the smaller test data set instead of the bigger reference data set took over 300 seconds to run compared to just above one second when not annotated. The *482.sphinx3* benchmark had similar issues.

The third problem occurred when running the *458.sjeng* benchmark. The problem there however was not the number of annotated functions but the fact that a function that gets called more than two million times (measured using callgrind) inside a loop has been selected for annotation. The end result was similar to the one of the *403.gcc* benchmark; the run took too long and had to be canceled. Moving the annotation one level up in the callgraph reduced the overhead significantly and ended up being under 1%.

While the approach of annotating as much as possible leads to the best security it comes with a high overhead for some applications. Doing performance measurements and

moving annotations from frequently called functions into the caller can greatly reduce the performance overhead as demonstrated with the *458.sjeng* benchmark.

User input annotation

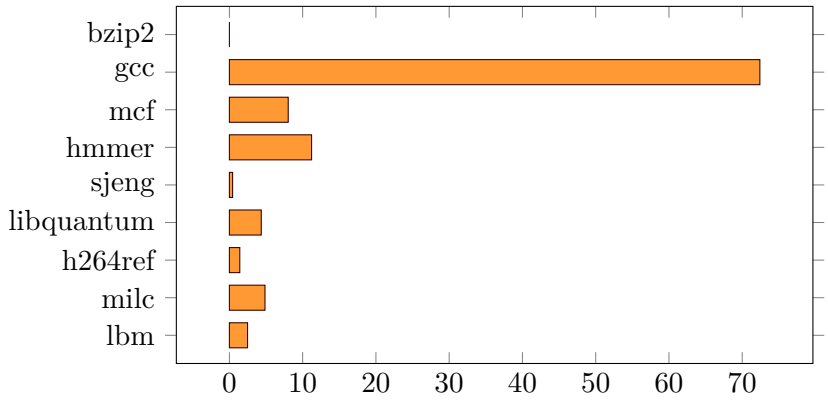

Figure 5.3: Performance overhead in % - User input annotation

Figure 5.3 shows the performance overhead of the user input annotation runs. The results vary between the benchmarks while some like *401.bzip2* and *458.sjeng* hardly show any overhead, the *403.gcc* benchmark shows a rather high overhead of 72.41%. The latter can be explained by the fact that gcc consists of a lot of small functions, which results into the function call overhead adding up. This also prevents the functions from being inlined by the compiler. All of the overhead measured in the full annotation scenario of the *401.bzip2* benchmark got eliminated in that configuration.

Reducing the set of functions to just the ones that process user input did not solve the performance problems observed with the *482.sphinx3* benchmarks in the full annotation scenario. There were still a few functions that got called frequently inside loops. As a consequence the benchmark is not included in the results, because the run had to be canceled.

5.2.4 Conclusion

Table 5.2 summarizes the benchmark results. As noted in the results the *458.sjeng* full annotation run has been done after moving one annotation into the caller. The overall results are mixed. Simply annotating the main function can be done automatically without expecting any performance overhead. The measured median overhead is as low as *0.21%*. Annotating all functions that use system calls without considering the context in which the functions get called can lead to very high performance degradation when frequently called functions end up being annotated. There are cases however where this leads to good results. For instance the *464.h264ref* benchmark only suffered 3.5 percent degradation despite being a computationally expensive benchmark. This would protect

Benchmark	Only Main	User Input	Full
401.bzip2	0.00 %	0.00 %	83.18 %
403.gcc	0.00 %	72.41 %	-
429.mcf	0.67 %	8.03 %	16.05 %
445.gobmk	0.00 %	-	-
456.hmmer	0.00 %	11.22 %	75.85 %
458.sjeng	0.44 %	0.44 %	0.67* %
462.libquantum	4.35 %	4.35 %	9.51 %
464.h264ref	0.21 %	1.24 %	3.51 %
433.milc	4.86 %	4.68 %	6.49 %
470.lbm	2.17 %	2.48 %	2.17 %
482.sphinx3	0.00 %	-	-

Table 5.2: Benchmark results summary

the user against malicious video files that try to exploit bugs in the video encoder with a limited effect on encoding times. Limiting the annotations to functions that only process user inputs reduced the overall overhead significantly. For instance the overhead for *401.bzip2* got reduced from over 80% to non-measurable. This leads to the next point. For some benchmarks there is only a negligible difference between annotating only the main function and annotating all functions that process user input. Overall most of the performance overhead is attributed to the thread creation when calling a sandboxed function. Annotating functions that consists of a few lines of code that would have been inlined by the compiler end up not being inlined after getting annotated. This means that it is preferable to annotate only functions that do not get called frequently like for instance inside a long running loop. For those functions the annotations should be moved one level up in the call graph to the caller. As demonstrated with the *458.sjeng* benchmark the overhead can be reduced significantly by doing that. Functions that do not get called frequently like for instance a function that parses a configuration file can be safely annotated without expecting any performance degradation.

CHAPTER 6

Summary and Conclusion

The solution presented in this thesis, Straightjacket, is based on the seccomp framework, which allows a user space program to load a system call filter into the kernel. The filter can then either allow or deny a specific system call. In addition to just checking the system call the filter can access the system call's arguments and validate them as well. The policy which defines which system calls are allowed is supplied by the application developer via annotations in the source code. The annotations apply to a specific C function. The Straightjacket code generator parses those annotations and generates code that applies the policy at run time.

Because seccomp filters act on processes every annotated function is modified to spawn a new thread, which is technically implemented as a process that has access to the parent's virtual memory. Once a filter is loaded for a process and the *PR_SET_NO_NEW_PRIVS* flag is set neither the process nor any child process it spawns can relax the restrictions applied by the filter. It can however tighten them further. By using that technique different functions can have more restricted privileges than the whole program. By applying annotations to the main function restricting the privileges of the whole program can be achieved as well.

There are two ways on how annotations can be used to restrict system calls. One way is to simply blacklist system calls that are known to be used by attackers but are not required for running the specific task. For instance system calls like *execve*, *connect* or *bind* which are often used by exploits to open local or remote command shells. The other more restricted mode allows supplying a whitelist of system calls. That way any system call not required for doing the function's task is denied. This follows principle of the least privilege but has some practical problems. In order for it to work the developer has to know which system calls a particular function requires. To aid the developer in finding the list of the required system calls, Straightjacket includes a static analyzer that can be used to identify system calls that a function needs. The blacklist mode is better suited for

functions that attempt to run arbitrary external code like interpreters. Also, because the filters apply to all child processes developers have to be careful when restricting system calls for a function that forks and executes other programs, because the filter would apply for them as well. On the other hand this can be used to restrict what those spawned programs can do, for instance limit their actions to the file descriptors of a pipe and disallow access to any other files. That's the reason why the two modes got implemented; it helps the developer to choose the appropriate level of security without limiting practicality.

By delegating the generation of the filters to the *libseccomp* library the solution isn't restricted to a particular processor architecture. System calls are identified by unique numbers. Those numbers can differ even between similar architectures like x86 and x86_64. The *libseccomp* library offers an abstraction to hide those architecture specific details.

To improve usability the Straightjacket tools are designed in a way that allows easy integration into existing build systems. That means that both the code generator and the static analyzer can be integrated into an existing build system with minimal effort. In most cases it is sufficient to change one or two variables in the Makefile. In addition to that it does not require any operating system modifications. It runs on any Linux system whose kernel is compiled with seccomp support.

The evaluation has shown that Straightjacket blocks common exploitation attempts successfully in most cases. The effectiveness is determined by the system calls available to the vulnerable function. It does protect against vulnerabilities inside the main function, which requires the whole set of system calls the application needs, but does not offer any additional protection compared to a traditional sandbox in that case. However, it has other advantages in that scenario. The sandboxing is completely inside the application and is transparent to the user. No supervisor daemon has to run neither is any external policy required. The performance overhead for just applying restrictions to the main function has been measured to be as low as 0.21% on average. Overusing the annotations, like annotating every function, can however result into a high performance overhead. This can be avoided by moving annotations out of frequently called functions one level up to the caller to reach an acceptable compromise between security and performance. It has been demonstrated that this approach can greatly reduce the performance overhead.

Straightjacket was not designed to replace existing security measures like DAC, MAC and various protections like stack canaries and address space layout randomization, but to supplement them and thus provide an additional layer of protection. For instance even if the application runs inside another sandbox, like a docker container, it can be further confined by restricting the exposed system calls to parts of the program that deal with untrusted input, which lowers the likelihood that an attacker, that managed to exploit the application, can find a way to escape out of the sandbox by attacking the host kernel.

CHAPTER 7

Further work

This thesis has demonstrated that the approach of an annotation based seccomp sandbox effectively blocks common exploitation attempts while being transparent to the user without requiring rewriting the application's source code by the developer.

The supplied tools can be further improved though. The code generator works as an additional compiler. This has the advantage of being compiler agnostic at the cost of an extra compilation step. The annotation syntax is designed in a way to fit into a compiler extension in form of a plugin. A similar approach is done for instance by OpenMP which also uses the pragma directive to allow the user to annotate code sections that should run in parallel to enhance performance. [60]

The run time enforcement code uses the *libseccomp* library for generating the seccomp-bpf filters. The filters could be generated at compile time and integrated into the resulting program in form of a binary string to eliminate the need of the library and reduce the initial setup cost. Similarly, the need of the Posix threading library could be eliminated by generating code that does the thread creation and waiting using system calls directly. However, because the library is available on all systems the duplication effort is probably not justifiable. Nevertheless, the possibility of not depending on any run time library exists and could be explored.

The static analyzer works on the source code of the application and therefore cannot identify system calls in some scenarios like the usage of external libraries and dynamic code generation. Extracting system calls from library functions is possible by analyzing the library files. To identify system calls used by dynamic code generation a dynamic analyzer can be used in addition to the static analyzer. Combing the data from both should lead to better results and enhance the usability of the tools by minimizing the manual effort required by the developers. Another limitation of the static analyzer is that it only identifies system calls. However, it could be extended to detect static arguments

of the system calls like for instance write to standard output to further improve the security of the automatically annotated functions.

The evaluation has shown that the security gained depends on the set of system calls a function requires for operation. A function that requires a large set of system calls simply provides a bigger attack surface. The static analyzer could be improved to detect such functions and propose re-factoring solutions to split the work into smaller functions that could be annotated to reduce the set of available system calls accessible by an attacker. Another possibility would be to allow annotating code blocks like for instance loops or specific branches of conditional blocks.

Another area that could be improved is the performance overhead. The combination of the static analyzer and the automated annotator developed as part of this thesis do not take performance into account. By doing automated run time profiling one can use the data from that profiling to tweak the annotations in a way to provide the best trade-off between performance and security. This approach would be similar to profile guided optimization (PGO) which is implemented in compilers like gcc. The basic idea is to first compile the application with additional instrumentation code and then run typical scenarios to generate performance statistics. The collected data is then used by the compiler generate code that optimizes for those cases. [61] This approach could be combined with re-implementing the code generator as a compiler plugin to reuse the PGO infrastructure available in contemporary compilers. Detecting potential performance bottlenecks due to overuse of annotations at compile time and warning the developer could be implemented to help adding the right amount of annotations.

While Straightjacket has been implemented for the C programming language the concepts can be reused for other languages. Adding support for a language like C++, which is similar to C should be possible by adapting the existing implementation. It has to be extended to handle C++ specific features like classes, virtual symbol tables and callbacks. The static analyzer would have to be adapted to support those features as well. Adding support for interpreted languages would require a re-implementation. Those are mostly high level languages that do not suffer from memory corruption issues at the same extent like lower level languages such as C or C++. Nevertheless a reduced set of system calls would still contribute to enhancing the security of programs written in those languages. Such an implementation could be done as part of an interpreter that generates the seccomp-bpf filters at run time. It could restrict system call arguments further than a compiler based approach by exercising knowledge of the current execution context.

Bibliography

[1] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[2] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[3] Michael Wikberg. Secure computing: Selinux. `http://www.tml.tkk.fi/Publications/C/25/papers/Wikberg_final.pdf`, 2007.

[4] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. Selinux and grsecurity: a side-by-side comparison of mandatory access control and access control list implementations, 2008.

[5] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[6] Andreas Gal, Christian W Probst, and Michael Franz. *A denial of service attack on the Java bytecode verifier*. Citeseer, 2003.

[7] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, page 6. ACM, 2011.

[8] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. Quantifiable run-time kernel attack surface reduction. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 212–234. Springer, 2014.

[9] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[10] Lukáš Aron and Petr Hanáček. Introduction to android 5 securit. In *Proceedings of Student Research Forum Papers and Posters at*, pages 103–112. CEUR-WS.org.

[11] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[12] Anthony Vérez and Guillaume Hugues. Security model of firefox os. 2013.

[13] Andrew Hoog and Katie Strzempka. *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier, 2011.

[14] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 2012.

[15] Hrushikesha Mohanty, MV Swamy, Pillalamarri Thilak, and Srinivasan Ramaswamy. Secured networking by sandboxing linux 2.6. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 3669–3674. IEEE, 2009.

[16] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306. Boston, MA, 2008.

[17] Clement Chong DeZhi, Lui YuYao, Teo ZhengLe, and Tan WeiJie. Comparing os security. *CS3235-Semester I, 2014-2015*, page 53.

[18] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: a trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 33–47. USENIX Association, 2014.

[19] Anthony A Aaby. Compiler construction using flex and bison. *Walla Walla College*, 2003.

[20] John Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[21] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *software, IEEE*, 19(1):42–51, 2002.

[22] William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, 2000.

[23] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[24] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[25] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The shellcoder's handbook*. Wiley Indianapolis, 2004.

[26] Serge E Hallyn and Andrew G Morgan. Linux capabilities: Making them work. In *Linux Symposium*, page 163, 2008.

[27] Tse Huong Choo et al. Trusted linux: A secure platform for hosting compartmented applications. *Enterprise Solutions*, pages 1–14, 2001.

[28] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, pages 29–46, 2010.

[29] Tomohiro Shioya, Yoshihiro Oyama, and Hideya Iwasaki. A sandbox with a dynamic policy based on execution contexts of applications. In *Advances in Computer Science–ASIAN 2007. Computer and Network Security*, pages 297–311. Springer, 2007.

[30] Zhenkai Liang, VN Venkatakrishnan, and R Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 182–191. IEEE, 2003.

[31] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*, pages 139–144, 2013.

[32] Vladimir Kiriansky, Derek Bruening, and Saman P Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, volume 92, 2002.

[33] Yusuf Motara and Barry Irwin. In-kernel cryptographic executable verification. In *Advances in Digital Forensics*, pages 303–313. Springer, 2005.

[34] Marco Slaviero, Jaco Kroon, and Martin S Olivier. Attacking signed binaries. In *ISSA*, pages 1–10, 2005.

[35] Mohan Rajagopalan, Matti A Hiltunen, Trevor Jim, and Richard D Schlichting. System call monitoring using authenticated system calls. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):216–229, 2006.

[36] Luke Dalessandro and Michael L Scott. Sandboxing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 171–180. ACM, 2012.

[37] Niels Provos. Improving host security with system call policies. In *USENIX Security*, volume 3, 2003.

[38] Aleksey Kurchuk and Angelos D Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *Security and Protection in Information Processing Systems*, pages 473–487. Springer, 2004.

[39] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. *Chakra vyuha (cv): a sandbox operating system environment for controlled execution of alien code.* IBM Thomas J. Watson Research Division, 1997.

[40] Kurt Natvig. Sandbox technology inside av scanners. *VIRUS*, 475, 2001.

[41] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[42] Yan Wen, Jinjing Zhao, Gang Zhao, Hua Chen, and Dongxia Wang. A survey of virtualization technologies focusing on untrusted code execution. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 378–383. IEEE, 2012.

[43] Faisal Al Ameiri and Khaled Salah. Evaluation of popular application sandboxing. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pages 358–362. IEEE, 2011.

[44] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[45] John Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25, 2014.

[46] Johannes Krude and Ulrike Meyer. A versatile code execution isolation framework with security first. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 1–10. ACM, 2013.

[47] Yoshihiro Oyama, Koichi Onoue, and Akinori Yonezawa. Speculative security checks in sandboxing systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[48] Ma Bo, Mu Dejun, Fan Wei, and Hu Wei. Improvements the seccomp sandbox based on pbe theory. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 323–328. IEEE, 2013.

[49] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, volume 3, pages 163–176, 2003.

[50] Zhen Li, Jun-Feng Tian, and Feng-Xian Wang. Sandbox system based on role and virtualization. In *Information Engineering and Electronic Commerce, 2009. IEEC'09. International Symposium on*, pages 342–346. IEEE, 2009.

[51] Robert NM Watson. Exploiting concurrency vulnerabilities in system call wrappers. *WOOT*, 7:1–8, 2007.

[52] Brian Greskamp, Pablo Montesinos, and Paul Sack. ipoj: User-space sandboxing for linux 2.4. *Urbana*, 51:61801.

[53] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*, 2015.

[55] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[56] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. *White Paper, Red Hat Inc*, 2003.

[57] Dejan Lukan. Miniupnpd analysis and exploitation. `https://www.viris.si/2013/12/miniupnpd-analysis-and-exploitation/?lang=en`, 2013. (Accessed: 2015-03-20).

[58] Frederic Perriot and Peter Szor. An analysis of the slapper worm exploit. *Symantec White Paper* `http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`, 2003.

[59] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-aslr on 64-bit linux. 2014.

[60] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[61] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile guided compiler optimizations. 2002.