BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATIC SECCOMP SYSCALL POLICY GENERATOR

AUTOMATICKÝ GENERÁTOR POLITIKY SYSTÉMOVÉHO VOLÁNÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MAREK TAMAŠKOVIČ**
AUTOR PRÁCE

SUPERVISOR **Ing. LENKA TUROŇOVÁ,**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

TAMAŠKOVIČ, Marek. *Automatic Seccomp Syscall Policy Generator*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lenka Turoňová,

Automatic Seccomp Syscall Policy Generator

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Tamaškovič
March 5, 2018

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

Chapter 1

Introduction

Nowadays, when malicious code or malware is becoming more and more sophisticated and pressing security risk, it is really needed to control a program behaviour and monitor what the program is doing in a system. Monitoring program behaviour can be done in many ways and one of the easiest ways is to use Intrusion Detection System (IDS). IDS is an out-of-the-box solution which can monitor i.e. where program wrote or read something and it is not allowed. After that, IDS is reporting this violation.

Another way is to monitor and block system calls (syscalls). Monitoring is performed using tools mentioned in the next chapter. The actual blocking can be performed with mandatory control access (MAC) (Apparmor, SELinux), sandboxing (seccomp) or others mechanisms. MAC refers to a type of access control by which the operating system constraints the ability of a subject or initiator to access or generally perform some sort of operation on an object or target. Seccomp is a Linux kernel module which allows to a process one-way transition to a secure state where the process can only use four syscalls. When the process tries to call another syscall then one of the four members set is terminated with SIGKILL. The set of allowed system calls can be extended using seccomp-bpf. This extension allows filtering system calls using a configurable policy implemented with Berkley Packet Filter (BPF) rules. This last part is an area on which I would like to focus in my thesis.

I aim to design and develop a tool which helps developers using libseccomp and seccomp-bpf. I plan to create policies for a specific program in a format readable by libseccomp or seccomp-bpf.

Chapter 2 describes syscalls and how to monitor them. In the next chapter of the thesis, I will illustrate how security facilities in Linux, such as systrace and seccomp, work. After the theoretical part, the design and development of a tool will follow. In conclusion, methodology how this tool was tested is described.

Chapter 2

System calls and Monitoring Tools

In this chapter, I will describe a term system call and make an overview of tools which can monitor the system calls. We will focus in detail on a strace tool which will be used as an input to my tool. The other applications are described briefly not as detailed as the strace tool.

2.1 System Calls

In computer terminology, the term syscall is a programmatic way in which a computer program requests a service from a kernel of the operating system on which is executed on. In other words, the system calls are functions used in the kernel itself. The system call appears to a standard developer as a C function call. This design is typical for monolithic kernels. We can find them on every UNIX system. The system calls are generated using interrupt, i.e., on Linux/i86 with an interrupt no. `0x80` or with system call named `syscall()` or `sysenter()` and these syscalls are handled by the kernel in a privileged mode. When a user invokes a system call, an execution flow is as follows:

- Each syscall is vectored through a stub in libc. Some syscalls are more complicated than others because of a variable length of the arguments, but the entry point and end point of syscall are still the same.
- In libc, the number of the syscall is then set to an `eax` register, and the stack frame is also set up.
- An interrupt number `0x80` is called and transferred to the kernel entry point. The entry point is the same for every system call.
- In the table of interrupts, a pointer to interrupt handler is found. After that, the execution of the interrupt handler follows which stores the content of the CPU registers and check if a valid syscall is called.
- The handler finds the corresponding offset in the table of interrupts `_sys_call_table`, where a pointer to syscall service is stored.
- Control is transferred to the syscall service.
- Syscall returns a value to the register `EAX` on 32-bit architecture or `RAX` on 64-bit architecture.

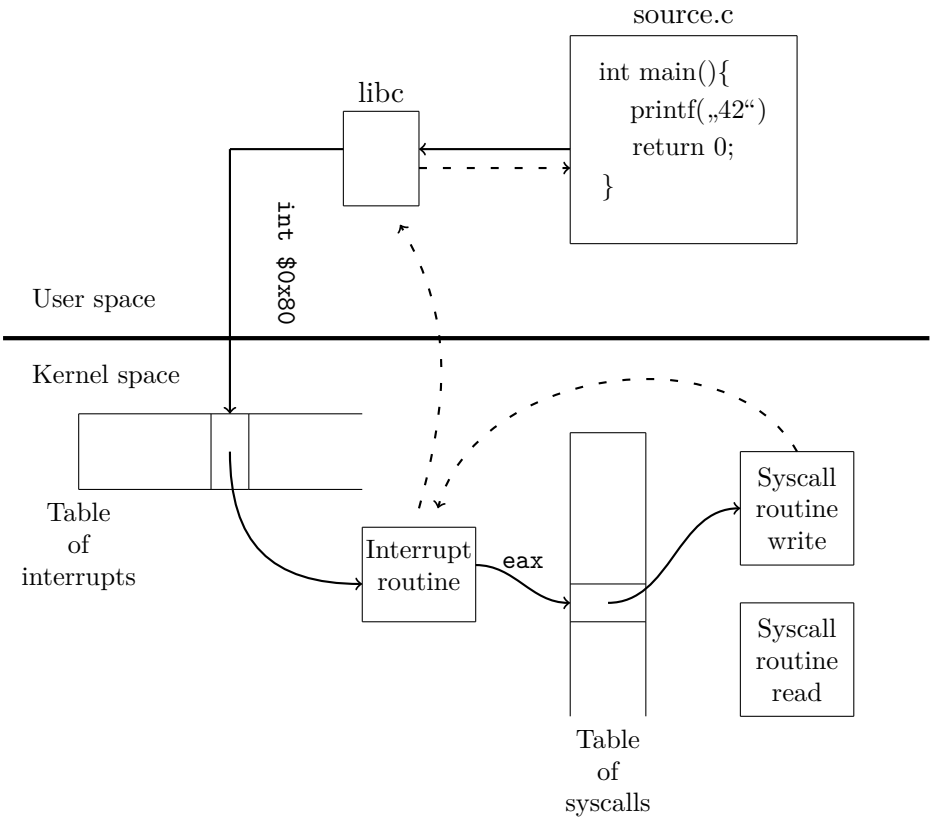

Figure 2.1: Interrupt handling in Linux

- At the end of the syscall, `_ret_from_sys_call` is called. This call is done before returning to userspace. It checks if the scheduler should be run, and if so, it calls it.
- Immediately after return from the system call to interrupt handler, `syscallX()` macro checks for a negative return value from the syscall, if so it puts a positive copy of a return value to a global variable `_errno`, for accessing from code like `perror()`.

This procedure is illustrated in figure 2.1 on page 5.

2.2 Monitoring

The most used and conventional method for monitoring is tracing, in other words watching what a program is doing during the execution. Tracing involves a specialized logging to record information, useful for debugging, about a program's execution. This can be done in multiple layers, from tracing which lines in the program was executed to individual instructions run on a CPU. Collecting this information can be done with various tools, i.e., strace, ftrace and a lots of more.

2.2.1 Strace

Strace[4] is a simple diagnostic, instructional and debugging tool. You can monitor every syscall or signal made by the program you are tracking. Using this tool is possible to log what an observing program demanded from the kernel. The individual recorded operations

can be, i.e., an attempt to open a file or delete a content of CPU caches. The strace also shows arguments for the called syscall. The developer can perform a fault injection for the specified set of syscalls as well, to simulate the program in faulty test cases. The next feature is that the Strace can trace child processes of the observing program. The log on the output will contain the system calls from the primary process and its child processes.

The main advantage of Strace tool is that it does not need any source codes. The observing program has not to be compiled with extra flags nor object files. Also, it does not matter if the application is statically or dynamically linked. This is useful because we only need executable binary. These features are helpful for my tool, but this will be more described in another chapter. Strace tool is straightforward, i.e., when one wants to run ls with strace he types in a command line:

```
>$ strace ls
```

In this case, the strace executes the `ls` command, and on the output, it occurs which system calls were called. An example of the strace output is in the next figure.

```
execve(„/usr/bin/ls", [„ls"], 0x7ffd0cf4ba60 /* 59 vars */) = 0
open(„/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, { st_mode=S_IFREG|0644, st_size=202163, ...}) = 0
mmap(NULL, 202163, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd781293000
close(3)
```

2.2.2 Ptrace

Strace is using ptrace[3] system call. The ptrace is used to implement debuggers and other tools for process monitoring. Basically, the strace call ptrace and attach to a tracee (monitored process). When the connection is established the tracee is halt before and after syscall. Now the tracer (strace) can observe and control the execution as well as inspect memory and registers of (tracee). With this information, strace can determine which syscall was called. With the second halt after syscall, the strace can get information of return value from syscall.

2.2.3 Ftrace

Ftrace[2] is an internal tracer which traces events in the kernel. It is designed for developers to examine kernel events. The main feature of this tool is to measure latencies and find issues that take place outside of the user-space. Ftrace is typically considered as a function tracker, but it is really a framework of several different tracing utilities. One neat feature of ftrace is measurement latency among interrupts, the lag between the time when the task is woken up and time when the task is scheduled in. Another frequent use of ftrace is event tracing. In the kernel is a massive amount of static event points that can be enabled with a tracefs file system. The event points provide an interface to observe what is going on in the various part of the kernel.

2.2.4 Dtrace

DTrace[1][12] (shortcut for Dynamic Tracing) is a performance analysis and troubleshooting tool. It is included in various operating systems, including FreeBSD, Mac OS, Solaris and Linux port is in development. This tool instruments all software, not just user-level software but also operating system kernel and device drivers. It supports dynamic tracing which means dynamically patching while running instructions with an instrumentation code. Static tracing is supported as well, but it needs to add tracepoints to the code. DTrace provides a scripting language called 'D' for writing scripts and one-liners. It is similar to C with AWK elements. With this script, you can create filters and summarize data in the kernel before passing to user-space. This design can decrease the overhead in performance sensitive systems.

For our purposes, DTrace is too complicated to setup or gather the information about syscalls. You need to write some scripts to define which syscalls you want to be informed with and in our use case, we need every system call.

2.2.5 SystemTap

SystemTap[6] is a tracing and probing tool that allows to gather information from probes injected into the kernel. It is similar to Dtrace but not so perfect. It started as a clone of Dtrace because it has uncompatible licence for using in GNU Linux. One of the common thing with Dtrace is that both tools use some type of scripting language. In this case is named 'SystemTap'. With this language you can specify what happens when some event occurs in the kernel.

SystemTap works as daemon which communicate with stap program. Stap is a small program that translates the SystemTap script to a kernel module. It is done in a few steps. At first it runs semantic analysis on the script. After that stap tries to translate it into a C code. The next step is to compile it as a kernel module and load it into the kernel. After load it is working and doing the useful part. When you send a signal to terminate the stap program it will unload the kernel module and stop working.

Similar as Dtrace the SystemTap is too complicated to work as system call monitor and it is not flawless. The Systemtap can't dereference the pointer address in system call but the strace tool can.

2.2.6 Autrace

The Linux Auditing System helps system administrators create an audit trail. Every action on workstation or server is logged into a log file. This tool can track security-relevant events, record the events and detect misuse or unauthorised activities by inspecting the audit log. You can also set which actions should or should not be reported.

Audit System is composed of two main parts. The first one *autrace* is kernel component which intercepts system calls, records event and sends these audit messages to the next part. The second component is audit daemon. This part of this tool is collecting the information emitted by kernel component. Emitted data is then stored as entries in a log file. As you can see this tool is not for monitoring one specific program, but it is designed to monitor the whole system. In the output is specified who and when executed the syscall, next is a current working directory, uid, gid, etc. Above specified functionality is useful for server administrators but not for our work.

Entry example in log file:

```
type=SYSCALL msg=audit(1434371271.277:135496):  arch=c000003e syscall=2
success=yes exit=3 a0=7fff0054e929 a1=0 a2=1fffffffffff0000 a3=7fff0054c390
items=1 ppid=6265pid=6266 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts0 ses=113 comm=„cat"
exe=„/usr/bin/cat" key=„sshconfigchange"
```

Chapter 3

Security Facilities in Linux

This chapter describes security facilities in GNU Linux operating system. Firstly we mention old tool named Systrace[16]. Later we will mention a secure computing module named Seccomp[14]. The next topic will be Berkley Packet Filter (BPF) because it is used in seccomp-bpf. The seccomp-bpf is an extension to basic seccomp. This extension can better describe the behavior of seccomp. In the end, there will be a description of libseccomp which is an easy to use library to the kernel syscall filtering.

3.1 Systrace

Systrace is security facility which limits an application's access to the system. It is similar to a newer tool named seccomp-bpf which is described later. The limitations are provided via system call blocking. The policy is generated interactively. Operations not covered by the defined policy raise the alarm. When an alarm is raised the user can refine the current policy. Systrace provides an option to generate policies automatically which can be immediately used in sandboxing. It is not flawless, so it sometimes needs minimal manual post-processing.

This tool provides cybersecurity by providing intrusion prevention. One of the uses is that you run systrace on the server. The systrace monitors all running daemons and can generate a warning when some incident occurs. These alerts can be sent to a system administrator and can provide information what happened.

3.2 Seccomp

A large number of syscalls are exposed to user space of a process. Many of this syscalls are unused for the entire lifetime of the process. This generates small possibility to misuse some syscalls to corrupt the process itself. A particular subset of applications benefits from a reduced set of syscalls by reducing exposed kernel surface to process. The filtering is done by seccomp. Seccomp filtering provides a means for a process to reduce the set of syscalls available to the process[7].

In most contemporary distribution, kernel module named Seccomp[14] is enabled. Seccomp stands for the shortcut of Secure Computing Mode. This module provides one way transition to a secure mode which restricts a thread to a few system calls `read()`, `write()`, `exit()`, `sigreturn()`. If the thread tries to call another system call then the one from the

four-member set, the whole process is terminated with signal `SIGTERM`. The drawback of this solution is that these four system calls are not enough for application to run correctly.

3.3 Berkeley Packet Filter and Seccomp

The seccomp filter mode that allows developers to write BPF programs that determine if a given syscall will be allowed or not. That allowance can be based on system call number or specific syscall argument values. Only the passed values are available, as any pointer are not dereferenced by the BPF. Filters can be installed using `seccomp()` or `prctl()`. The BPF program should be constructed first, then installed in the kernel. After that, every system call triggers the filter code. The installed filter cannot be removed or modified. Another property of applied filter is that the filter is inherited from a parent process to every child process when using `fork(2)` or `exec(2)`.

A BPF language came in 1992 for a program called tcpdump which is a monitoring tool for network packets. The volume of packet can be colossal, so it makes the transfer to user-space expensive. The BPF provides a way to do filtering in the kernel and the user space only handles those packets which was interested in.

The seccomp filter developers realised that they wanted very similar task. After that, the BPF was generalised to allow system call filtering. After the update, there is a tiny BPF virtual machine in the kernel space that interprets the BPF instructions.

The next update of BPF was to eBPF which stands for extended BPF. This update was released in Linux Kernel 3.18 for tracepoints later in 3.19 for raw sockets and in 4.1 for performance monitors. The eBPF brought the performance improvements and new capabilities.

The eBPF virtual machine is widely used in the kernel for various filtering:

- eXpress Data Path (XDP) *is a high performance, programmable network data path in the Linux Kernel*
- Traffic control
- Sockets
- Firewalling $xpf\_bpfmodule$
- Tracing
- Tracepoints
- kprobe *dynamic tracing of a kernel function call*
- cgroups

3.3.1 eBPF - Specification

The eBPF virtual machine has 64-bit RISC architecture designed for one to one mapping to 64-bit CPUs. Instructions are similar to classic BPF for simple conversion to eBPF. The old format had registers A and X instead of current 11 registers, grouped by function as described below [17].

- R0 exit value for eBPF
- R1 - R5 function call arguments to in-kernel functions
- R6 - R9 callee-saved registers preserved by in-kernel functions
- R10 stack frame pointer (read only)

So far 87 internal BPF instructions were implemented. Opcode field has room for new instructions. Some of them may use 16/24/32 byte encoding.

Same as the original BPF (the new format runs within controlled environment) is deterministic and the kernel can easily prove that. The safety of a program can be verified in two steps. First step does depth-first-search to forbid loops and CFG[1] validations. The second step starts from first instruction and descends all possible paths in CFG. It simulates execution of every instruction and examines the state of registers and stack [17].

3.3.2 eBPF - Instruction Encoding

An eBPF program is a sequence of 64-bit instructions. All eBPF instructions use the same design of instruction encoding which is shown in 3.1. As you can see in the figure, there are 5 parts that are opcode (operation code), dst (destination), src (source), offset, immediate [17].

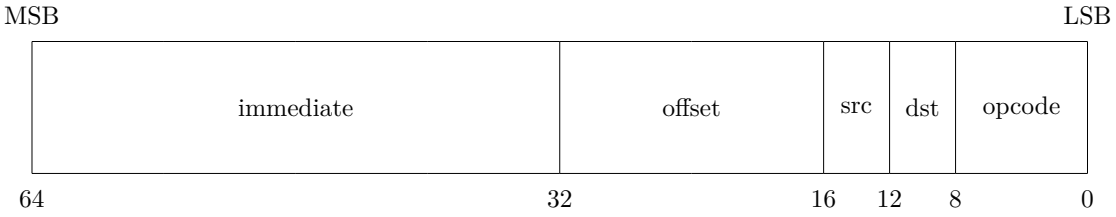

Figure 3.1: eBPF instruction encoding

3.4 Libseccomp

Libseccomp[15] is easy to use library which provides a platform-independent interface to the Linux Kernel's syscall filtering. The libseccomp API is designed to abstract an user from underlying BPF based syscall filtering and present a more conventional function-call based filtering interface. The interface should be more familiar to and quickly adopted by, application developers. The comparison of libseccomp and raw BPF filter is shown in figures A.1 and A.2.

One of the advantages of libseccomp is that it has a permissive mode in which every syscall violation is reported to the user. This feature can be helpful if you want to obtain information which syscalls was called. This use case is really similar to the syscall monitoring. But it is really tough to depend on this output because it is in development.

[1] Control flow graph

Chapter 4

Solution Design

This chapter will describe to the reader the solution design from architectural and behavioral point of view. It will show particular parts of solution, their architecture and issues.

4.1 Requirements

Specification of requirements for release:

1. Application will have only CLI[1].
2. Implementation of programming part will be in C++14[9].
3. Application will be designed with consideration of good OOP
4. Application will consist of these main parts:
   (a) Parser
   (b) Optimizer
   (c) Policy generator
   (d) Logger
5. Parser will be implemented using PEG[2][5] design.
6. Optimizer will have at least three optimizing methods:
   (a) Strict - No advanced methods used. Interpret internal structure.
   (b) Weak - Allow interval between minimum and maximum value found in set of arguments
   (c) Advanced - Combine above methods.
7. Policy will be generated with libseccomp[15] syntax as C language[10] code.
8. Logger will be able to log different log levels (debug, trace, info, error, warning)
9. Logger will log into user specified files or in predefined files located in current working directory.

[1] Command Line Interface
[2] Parsing Expression Grammar

4.2 Architecture

The architecture of this application is generalized using architectural patterns. In this case was used *Pipe-and-Filter*[11] architectural pattern. This pattern was chosen for the best description of a current problem. A big inspiration came from compilers. They are very similar to this application. They've got parser, optimizer, output generator as well and every component is dependent on component before. There are two main cases as shown in figure 4.1.

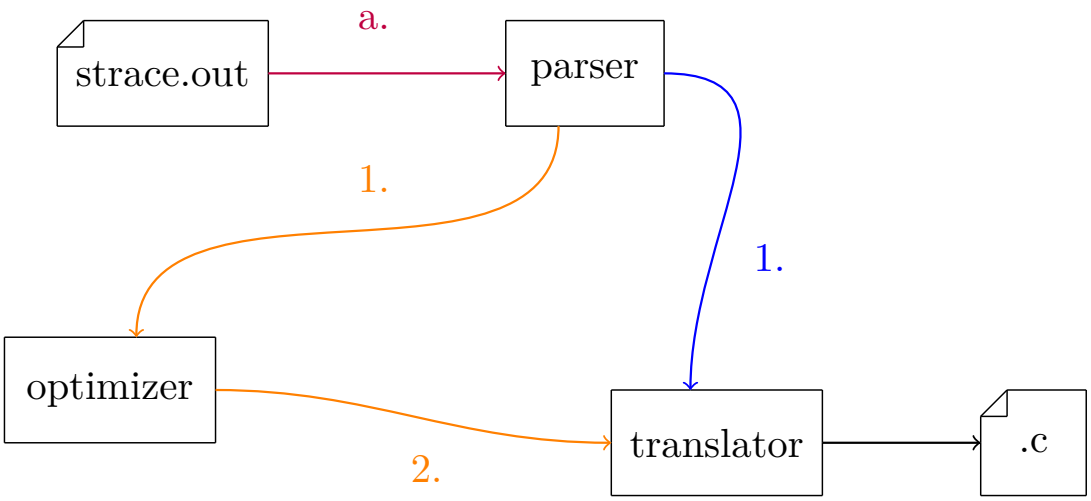

Figure 4.1: Architecture of strace2seccomp

4.2.1 Without Optimizer

In this case optimizer is not in the pipeline, a raw input is translated into libseccomp commands. This pipeline is shown in fig. 4.1 as a sequence of purple, blue and black arrows. There are no optimizations when optimizer is turned off.

The main problem of not using a optimizer is not proper working program. There is a way that any minor change in system call parameters can results into a program termination. In complex program is no way to be definitely sure if every syscall was caught. That can be a big problem in programs which uses seccomp. That is the main reason why is this option not recommended. But sometimes some users can be found which do not want optimizing the strace. It can be found only in small applications which has some limited functionality.

The main disadvantage of this solution is that it is too robust to place it in a code and is very strict. It can kill a program even in a false positive case when user change some of the parameters that was not covered in strace input files.

4.2.2 With Optimizer

In this case when optimizer is turned on *(path: purple, orange, black in fig. 4.1)*, we can specify which type of optimization we want. As mentioned in section 4.1, there are two optimization variants. Those variants can be switched with runtime arguments in CLI.

The pitfalls of this case is allowing program to continue even in inappropriate circumstances. Invalid circumstances can be defined as a bad syscall argument treated as a valid.

It can happen when you allow set of arguments for specific syscall. This is not secure however it is more suitable for work than the case without optimizer.

You can minimize these pitfalls by providing a lot of strace input files. The best case is when you provide strace files from every major complex test case (with big coverage).

4.2.3 Logger

One of the important part of every component is logging. With logging you can see what was the program doing in specific parts. The logger should log everything in a log file, print logs on standard output or both. It should handle various logging level e.g. debug, trace, info, warning or error. The logger must be accessible from every component. The output should be written in user specified log files instead of a system logs.

4.2.4 Parser

The parser is crucial part of the whole application because it will put everything in a IR(intermediate representation). As input to the parser is a output from Strace tool. The output was described in section 2.2.1. As you can see the output is in structured form and has a unambiguous syntax which means that no error should occur during the parsing part. When syntax error occur, the program should inform where the error is located in the input file. Next step should be a proper exit. Parser should have an option to identify all errors in the input file which can be helpful of identifying more errors on once. Another feature the parser will have is a printing structured parsed data.

4.2.5 Parsing Expression Grammar

Parsing Expression Grammar was introduced by Ford in 2004. PEG is a type of analytic formal grammar which describes a formal language in terms of a set of rules for recognizing strings in the language. This type of grammar is really similar to a top-down parsing languages. As well it looks very similar to context-free grammars.

Parser that parses the PEG is named a Packrat Parser. Packrat parser can be easily constructed for any language described by an LL(k) or LR(k) grammar, as well as for many languages that require unlimited lookahead and therefore are not LR. Packrat parsers ale also much simpler to construct than bottom-up LR parsers, making it practical to build them by hand. It can directly and efficiently implement common disambiguation rules such as *longest-match*, *followed-by*, and *not-followed-by*, which are difficult to express unambiguously in a context-free grammar or implement in conventional linear-time. Finally, both lexical and hierarchical analysis can be seamlessly integrated into a single unified packrat parser.

The main disadvantage of packrat parsing is memory consumption. The worst case asymptotic computational complexity is very similar to the conventional algorithms (linear in the size of the input).

The one of the many problems with right to left parsing algorithm is that it computes many results that are never needed. Other inconvenience is that we must carefully determine the order in which the results for a particular column are computed. *Packrat parsing* is a lazy derivation of a tabular algorithm that solves both of these problems. It computes results only when thet are needed, in the same order as the original recursive descent parser would. [8]

Additive	←	Multititve '+' Additive | Multitive
Multitive	←	Primary '*' Multitive | Primary
Primary	←	'(' Additive ')' | Decimal
Decimal	←	'0' | . . . | '9'

Table 4.1: Example of a grammar for a trivial language

4.3 Optimizer

Optimizer the main part of this tool. This part will be reducing the (intermediate data structure, IDS). There are three approaches of reducing IDS.

- **Strict** optimization is defined as 1:1 to strace input file. It means that it will every record in strace interpret as a strict rule. So only that one case is only possible to run. Every minor change in system call will kill the process. This strict way means for optimizer that it is turned off.
- **Weak** optimization is defined as finding minimum and maximum values on specific levels of IDS. It can be found on multiple levels as well.
- **Advanced** optimization is defined as combination of both strict and weak optimizations. In some specific cases it will use only the exact values and in other cases it will use weak optimizations. This combination should be more strict than the weak o. and weaker than the strict o.

Algorithm 1: Weak optimization

```
Input: intermediate data structure IDS
Output: list of rules rules
foreach syscall sc in the IDS do
    foreach argument arg in the syscall sc do
        num_arg ← get_num_args(p);
        for lvl ← 0 to num_arg do
            intervals.append(get_minmax(arg, lvl));
        end
        rules.append(sc, intervals);
    end
end
```

4.4 Intermediate Data Structure

One of the main parts of strace2seccomp is IDS in which are the individual system calls stored. The main idea of this ADT (Abstract data type) to be simple, readable with smallest redundancy possible. This can be done only with good abstraction and good design.

IDS is represented as a tree structure. In this structure are different information about syscall represented on different level in the tree. The root node in tree has child elements which represents a individual system calls. In this case we are naming these nodes as

(system call node, SCD). In SCD is stored information about system call number and it have multiple children. The nth level represents the nth argument of a specific system call. The whole system call with every argument can be read as a path from the root node to the leaf node. The IDS representation is shown in figure 4.2.

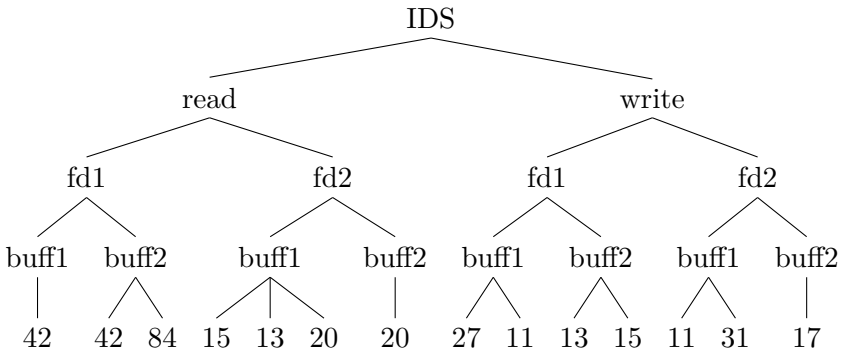

Figure 4.2: Visualized IDS as a tree

Chapter 5

Development of strace2seccomp

5.1 Logger

There are many ways how can logger be implemented. One of the first ideas were implement custom logger. This idea wasn't bad but it will be a reinventing wheel. Another way is using the logger implemented in boost library[1]. I decided to use this library because it was audited and tested by community, is widely supported and is well documented.

5.2 Input

5.3 Intermediate Representation

5.4 Output

5.5 Heuristics and Optimizations

5.5.1 Minimax

5.5.2 Strict

5.5.3 Smart

[1] http://www.boost.org/doc/libs/1_63_0/libs/log/doc/html/index.html

Chapter 6

Software Verification

This chapter will describe activities used to assure quality control of the developed tool. Firstly I want to introduce on which aspects we will focus. One of the aspects is *module testing*. The main purpose of module testing is to detect errors in submodules, in communication among them, in passing data through data structures. Another aspect of verification is *system testing* merged with *acceptance testing*. In this type of testing we will check if the strace2seccomp tool has valid architectural design. Next aspect which will be checked is *static analysis*. Static analysis is type of testing which does not requires run the program but requires a source code of the tool. The static analyzer will analyse the source codes with different heuristics and produces a analysis of detected errors.

6.1 Module Testing

Module testing is part of whole quality control process. This testing can provide us how functional are particular components and if they meet the requirements. Table 6.1 shows us the description of module's test suits.

Module / Component	**Test descriptiom**
Params	Validity of recognized runtime arguments
StraceParser	Syntax testing, validity of parsed data, correct error handling

Table 6.1: Test plan

6.1.1 Params Testing

about using params class in custom test set and manualy check if the correct flags was ommited.

6.1.2 StraceParser Testing

StraceParser module is responsible for parsing the strace output and translates it to intermediate data structure. Testing of this module can be done with various techniques. First one which is used is fuzzy testing or fuzzing described in section above 6.3.1. AFL fuzzer was used in the process.

6.2 System and Acceptance Testing

6.3 Testing Methods

6.3.1 Fuzzing

The term fuzzing was first used by professor Barton Miller who used fuzzing to test robustness of UNIX applications in 1989 [18][13].

6.3.2 American Fuzzy Lop

6.3.3 HongFuzz

6.3.4 Radamsa

6.3.5 Bunny the Fuzzer

6.4 Static analysis

6.5 Test Requirements

6.6 Results

6.7 Test Statistics

Bibliography

[1] *dtrace(1) Linux User's Manual*. November 2017. version *3.1-5.fc26*.

[2] *ftrace(1) Linux User's Manual*. November 2017. version *0.4-56.fc26*.

[3] *ptrace(2) Linux User's Manual*. November 2017. version *4.09*.

[4] *strace(1) Linux User's Manual*. November 2017. version *4.19*.

[5] A. Moss: *Derivatives of parsing expression grammars. Electronic Proceedings in Theoretical Computer Science, EPTCS*. vol. 2. 2017: pp. 180–194. ISSN 20752180. doi:{10.4204/EPTCS.252.18}.

[6] Domingo, D.. Cohen, W.: *SystemTap 3.0*. [Online, accessed 21.2.2018].
URL: https://sourceware.org/systemtap/SystemTap_Beginners_Guide.pdf

[7] Drewry, W.: *SECure COMPuting with filters*. [Online, accessed 27.11.2017].
URL: https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

[8] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl. *SIGPLAN Not.*. vol. 37, no. 9. September 2002: pp. 36–47. ISSN 0362-1340. doi:10.1145/583852.581483.
URL: http://doi.acm.org/10.1145/583852.581483

[9] Information technology - Programming languages - C++. Standard. International Organization for Standardization. Geneva, CH. March 2014.

[10] Information technology - Programming languages - C. Standard. International Organization for Standardization. Geneva, CH. March 2011.

[11] J. Andrés Díaz-Pace and Marcelo, R, Campo: *ArchMatE: from architectural styles to object-oriented models through exploratory tool support. ACM SIGPLAN Notices*. vol. 40. 2005: page 117. ISSN 03621340. doi:{10.1145/1103845.1094821}.

[12] Leventhal, A.. et al.: *About DTrace*. [Online, accessed 2.10.2017].
URL: http://dtrace.org/blogs/about/

[13] Marhefka, M.: Automatizované fuzz testování aplikací komunikujících přes systém D-Bus. 2013.
URL: http://hdl.handle.net/11012/55032

[14] markus@chromium.org: *seccompsandbox - overview.wiki*. [Online, accessed 2.10.2017].
URL:
https://code.google.com/archive/p/seccompsandbox/wikis/overview.wiki

[15] Moore, P.: *Libseccomp*. [Online, accessed 30.11.2017].
URL: https://github.com/seccomp/libseccomp

[16] Provos, N.: *Systrace - Interactive Policy Generation for System Calls*. [Online, accessed 2.10.2017].
URL: http://www.citi.umich.edu/u/provos/systrace/

[17] Schulist, J.. Borkmann, D.. Starovoitov, A.: *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. [Online, accessed 11.12.2017].
URL: https://www.kernel.org/doc/Documentation/networking/filter.txt

[18] Takanen, A.. DeMott, J.. Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc.. first edition. 2008. ISBN 1596932147, 9781596932142.

Appendix A

Comparison of libseccomp and raw BPF filtering

A.1 BPF

```
1 int myapp_seccomp_raw_start(void)
2 {
3 struct sock_filter filter[] = {
4   BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 4),
5   BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 0x00, 0x12),
6   BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 0),
7   BPF_STMT(BPF_JMP+BPF_JGE+BPF_K, 0x40000000, 0x10, 0x00),
8   BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_open , 0x0e, 0x00),
9   BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_close, 0x0d, 0x00),
10  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, __NR_read, 0x00, 0x0d),
11  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 20),
12  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x0b),
13  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 16),
14  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x09),
15  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 28),
16  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x00, 0x02),
17  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 24),
18  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, 0, 0x05, 0x00),
19  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 36),
20  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, (SSIZE_MAX >> 32), 0x00, 0x02),
21  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 32),
22  BPF_STMT(BPF_JMP+BPF_JEQ+BPF_K, (SSIZE_MAX & 0xffffffff), 0x01, 0x00),
23  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
24  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
25 };
26 struct sock_fprog prog = {
27   .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
28   .filter = filter,
29 };
30 if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) < 0)
31   return -errno;
32 if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) < 0)
33   return -errno;
34 return 0;
35 }
```

Listing A.1: Using raw BPF filtering

A.2 libseccomp

```
1 int myapp_libseccomp_start(void)
2 {
3   int rc;
4   scmp_filter_ctx ctx;
5   ctx = seccomp_init(SCMP_ACT_KILL);
6
7   if (ctx == NULL)
8     return -ENOMEM;
9
10  rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(open), 0);
11
12  if (rc < 0)
13    goto out;
14
15  rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
16
17  if (rc < 0)
18    goto out;
19
20  rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 3, SCMP_A0(
      SCMP_CMP_EQ, STDIN_FILENO), SCMP_A1(SCMP_CMP_NE, 0x0),SCMP_A2(SCMP_CMP_LT,
       SSIZE_MAX));
21
22  if (rc < 0)
23    goto out;
24
25  rc = seccomp_load(ctx);
26
27 out:
28  seccomp_release(ctx);
29  return rc;
30 }
```

Listing A.2: Using simpler libseccomp wrapper

Appendix B

other appendix