Seccomp Filters from Fuzzing*

Marcus Gelderie, Valentin Barth, Maximilian Luff and Julian Birami
Aalen University of Applied Sciences, Beethovenstraße 1, Aalen, Germany

Keywords: Fuzzing, Seccomp, Sandbox, Dynamic Analysis.

Abstract: Seccomp is an integral part of Linux sandboxes, but intimate knowledge of the required syscalls of a program are required. We present a fuzzer-based dynamic approach to auto-generate seccomp filters that permit only the required syscalls. In our model, a syscall is required, if any execution path leads to its invocation. Our implementation combines a symbolic execution step and a custom mutator to take command line flags into account and achieve a large coverage of the SUT. We provide an evaluation of our tool on popular command line tools and find up to 100% of the system calls found through manual analysis.

1 INTRODUCTION

Sandboxing is an integral part of many modern security concepts, especially when third-party software of unknown quality is involved. As such, sandboxes can help mitigating supply chain risks to some extent. Sandboxes leverage mechanisms provided by the OS to implement their functionality. Seccomp[1] is an example of such a facility in the Linux kernel. It is a common component of many Linux-based sandboxes, such as *Bubblewrap*[2] or *Firejail*[3] and many container runtimes, such as LXC[4] or `runc`[5]. Using Seccomp, a sandbox can define *filters* that the kernel consults to disallow specific syscalls for a process and its offspring. However, definition of clear seccomp rules is challenging. When software is pulled in through the supply chain, intimate knowledge of that software is rare and usually unwanted, because it would defeat the purpose of outsourcing development in the first place. Writing a secure yet functional filter is therefore far from trivial.

Common strategies like over-approximations and manual test-driven seccomp filter generation neglect the least privilege principle or do not scale. Therefore we propose a novel approach to automatically generate seccomp filters in this paper. We propose to use fuzzers in order to trigger syscalls inside a program. Specifically, we provide an architecture and proof-of-concept implementation of a system that automatically provokes and identifies syscalls within a system under test (SUT) and creates a seccomp filter for that SUT which only allows precisely those syscalls that were observed. We demonstrate the effectiveness of this approach by generating filters for some commonplace software. Finally, we reflect critically on the situations in which a fuzzer-based filter generation can be of value. Our tool is available on GitLab[6].

1.1 Related Work

Auto-generation of seccomp filters from source code has been studied before, e.g. (Canella et al., 2021). Their solution is primarily based on static analysis with an optional dynamic component. However, this dynamic component serves to augment static analysis, particularly if the SUT forks and executes further binaries. Other approaches to automatic generation of filters exist (DeMarinis et al., 2020; Ghavamnia et al., 2020). All these approaches are entirely or predominantly based on static analysis. As such, they suffer drawbacks common to static analysis, such as scalability and dependency management. In particular, static approaches often fail to find system calls (syscalls) that are made through dynamic dependencies, such as `libc`, which is, arguably the common

*This Work Was Supported in Part by a Grant from the Kessler + Co Foundation (Stiftung KESSLER+CO Für Bildung Und Kultur).

[1]https://www.kernel.org/doc/html/v5.16/userspace-api/seccomp_filter.html

[2]https://github.com/containers/bubblewrap

[3]https://github.com/netblue30/firejail

[4]https://github.com/lxc/lxc

[5]https://github.com/opencontainers/runc/

[6]https://gitlab.com/iot-aalen/fuzz-seccomp-filter

Gelderie, M., Barth, V., Luff, M. and Birami, J.
Seccomp Filters from Fuzzing.
DOI: 10.5220/0011145100003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 507-512
ISBN: 978-989-758-590-6; ISSN: 2184-7711
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

case.

By contrast, (Lopes et al., 2020) approaches the problem of syscall detection from a dynamic perspective. The authors leverage unit tests and fuzzing combined with dynamic syscall detection. Their focus is on integration with a CI pipeline and on detecting syscalls within the narrow parameters of deployment. As such, they assume that test coverage will usually cover all syscalls and use fuzzing only as a fallback. The fuzzing solution outlined in (Lopes et al., 2020) is therefore not very sophisticated. By contrast, our proposed system combines symbolic execution and fuzzing. It analyzes the binary to detect CLI flags and involves these findings in the fuzzing process to determine all *possible* syscalls.

2 PRELIMINARIES

AFL++ (Fioraldi et al., 2020) offers different custom mutator hooks to replace or complement its integrated methods. We briefly mention the hooks relevant to this paper. In the `init` method of AFL++, the Custom Mutator itself is initialized, as well as the PRNG used for fuzzing. With the `fuzz` hook the initial fuzzing procedure is determined. This method is called until the threshold fuzz_count is reached. Non-deterministic tweaks can be defined within the `havoc_mutation` method. Lastly, the result of all performed steps is handed over to `post_process` – for example to restore a valid file format.

In AFL the fuzzing strategies for mutating a given input based on different patterns are called tweaks. In the havoc mutation cycle these tweaks are stacked when all deterministic strategies are already used. For example these tweaks could be block duplications or deletions. [7]

3 GENERATING FILTERS WITH FUZZING

3.1 Problem Statement

Our goal is to generate a seccomp BPF filter that permits only those syscalls that are absolutely necessary for the given program to function correctly. More specifically, we aim to identify *all* syscalls that the program might execute along *any* of its execution paths. Formally, given a program P, we say that a syscall is *required* by P if there exists an input I and an execution of P on I that witnesses this syscall. The set of all syscalls required by a program is denoted by $\mathcal{S}_P$. Note that $\mathcal{S}_P$ is an over approximation of the syscalls that are actually performed by P during any of its executions. For instance, if $\mathcal{S}_P$ is a program that optionally connects to a Unix domain socket, the `socket()` syscall is present in $\mathcal{S}_P$, but may never be per called in practice if P is not configured in this way.

The set $\mathcal{S}_P$ depends on the input given to P. In practice, this input consists of command-line arguments, environment variables, standard input, input read from files (in particular config files), and IPC. In the present paper, we restrict ourselves to standard input and command-line arguments. We assume that we have access to the source code of P.

Our goal is to automatically generate an approximation A of $\mathcal{S}_P$, such that $A \subseteq \mathcal{S}_P$ and, ideally, $E = \mathcal{S}_P \setminus A = \emptyset$. The size of the set difference E serves as quality criterion. Note that since we require $A \subseteq \mathcal{S}_P$ we cannot allow the detection of false positives. False negatives are allowed. Quality is measured in the number of false negatives, with lower numbers being better.

3.2 Architecture and Overview

To find syscalls in a given program, we fuzz the target in order to find new execution paths and thereby trigger new syscalls. Secondly, we need to detect those syscalls, and, finally, generate the corresponding seccomp filter.

To perform these tasks, we use AFL++ (Fioraldi et al., 2020) as fuzzer for the target. We implement a Custom Mutator to fuzz commandline arguments separately. This way, branches will be reached more reliably in a reasonably amount of time. Secondly, we rely on the seccomp library [8] to log used syscalls. Furthermore, it enables us to crash the program for untracked syscalls, which is a detectable behavioural change for AFL++. These test cases will be saved as crashing input. Seccomp filters are added prior to fuzzing by a dedicated *wrapper* program which then proceeds to `exec` the SUT. Lastly, the Linux audit framework (Zeng et al., 2015) was used to check if found syscalls were originating in the SUT or the wrapper with its AFL++ specific instrumentation.

The high level architecture of components is displayed in figure 1. The colour coding shows the existing components in red, our developed components in green as well as a pre-evaluation step in blue. In the following we give a brief description of the overall architecture. Further below we explain the more

[7] https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

[8] https://github.com/seccomp/libseccomp

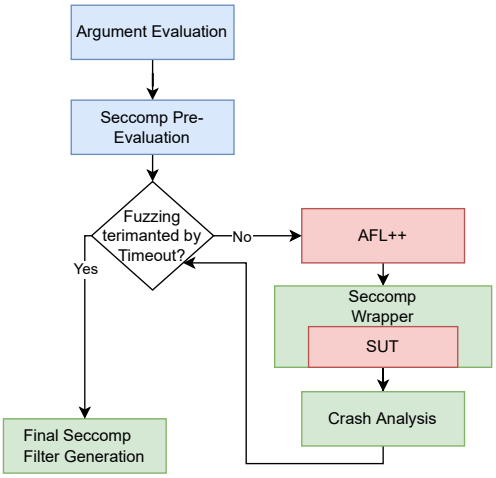

Figure 1: High level Architecture.

involved steps, such as custom mutation, in greater detail.

The detection engine performs the following steps as shown in figure fig. 1). Sometimes a step is repeated multiple times, as indicated. In the pre-evaluation phase the symbolic execution engine determines valid CLI flags and a syscall baseline is created by executing the SUT with the found CLI flags.

In the main cycle the CLI flags and stdin are stored in binary format in corpus files and mutated. They are then handed over to a Seccomp Wrapper, which retrieves the CLI flags from stdin to create a corresponding `argv`. Further, it removes CLI flags and then hands over stdin to the SUT. It also invokes the SUT after establishing seccomp rules to terminate on undiscovered syscalls. AFL++ halts on each crash, because its likely indicative of a new syscall being found. The list of found syscalls is then updated accordingly and fuzzing resumed at the crashing queue entry. Lastly, after a configurable time has elapsed without detecting a new syscall the fuzzing process is not restarted and the seccomp filter is output.

3.3 Implementation

Pre-evaluation: First an analysis step is performed to gather arguments of the SUT. This step utilizes the found strings in the binary as well as the symbolic execution engine KLEE (Rizzi et al., 2016) to generate a list of valid CLI flags, which is referred to as Argument Evaluation. Afterwards the binary is just executed with the gathered CLI flags and the same techniques later on mentioned, like establishing seccomp filters with a wrapper, are applied. Thereby a first syscall baseline is gathered.

Lastly, the first flag, which does not crash the binary, is used as the initial test corpus. During fuzzing, AFL++ actually fuzzes a wrapper, described below, that is responsible for loading seccomp filters, handing over the input test corpus, and invoking the SUT with the right CLI flags.

Seccomp Wrapper: Originally AFL++ is not structure-aware and furthermore only fuzzes standard input or file inputs. To deal with that challenge the argument fuzzer inline header[9], provided by AFL, was adjusted to only read the first 1kByte of standard input (instead of everything) and assign them as argument vector. The rest is treated as standard input. Corpus files follow this format: The first kByte represents the argument vector, the rest is treated as standard input and would correspond to a traditional corpus file.

Another task for the seccomp wrapper is to load the list of syscalls detected so far and parse it to generate the seccomp filter before invoking the SUT. To pass the binary check at the beginning of the fuzzing process, a few syscalls must be allowed manually. These syscalls are `shmat` and `shmctl`, which are needed for AFL to exchange coverage information and `execve` to invoke the SUT itself.

AFL++ Custom Mutator and Post Processing: Although the schedule, trimming cycle and overall structure of AFL++ are not changed, a custom mutator was added to take the list of arguments from the symbolic execution into account and to avoid wasting cycles on unlikely CLI flags (non-printable characters, for instance). The mutator itself is divided into a `fuzz` method, which serves as the initial step, the `havoc` method of AFL++, which is used periodically, but not always, and the `post_process` method, which is always called after all mutations have been performed. The `fuzz` method probabilistically chooses works in one of two ways: It either randomly chooses an argument vector from the list of valid CLI flag combinations (e.g. `-f --verbose`), *or* generates valid-looking CLI flags from a random number generator (e.g. `--dsfY -T`). When the havoc mutation is performed, the entire corpus file, including the vector contained in the first 1K, undergoes the *stacked tweaks* mutations (e.g. bitflips), which are less suitable for CLI flags. We therefore set a boolean flag in the custom `havoc` method. This flag is evaluated in the `post_process` hook. If set, this hook will choose a random argument vector from the valid list of CLI flags. However, if the flag is unset `post_process` does not alter its input. This is done to avoid un-

[9]https://github.com/google/AFL/blob/master/experimental/argv_fuzzing/argv-fuzz-inl.h

wanted changes to the output of the `fuzz` method, which is the only way to detect new CLI flags after the symbolic execution step at the beginning.

Fuzzing Process and Crash Analysis: The fuzzing and analysis procedure is displayed in figure 2. AFL++ first checks whether there already is an output directory and, in that case, resumes the fuzzing process, which is possible because the queue of input corpora is the only state saved by the tool (Böhme et al., 2019). At that stage AFL++ is also instructed to bench the binary until the first unique crash happens.

It will then hand the generated test cases over to the seccomp wrapper, which is going to execute the SUT as mentioned. Every time the fuzzing process is interrupted it is checked if the stop was because fuzzing time has elapsed. In the event of an actual crash, the input corpus is minimized with `afl-tmin`. This step is not necessary, but it helps significantly to reproduce the test case for manual analysis purposes.

Then, an Audisp plugin is activated that logs seccomp messages to another file, guaranteeing that our tool does not have to parse the complete audit log file each time. We assume that the system does not produce any seccomp messages resulting from other binaries in that time frame.

Afterwards, the test case is executed once again and, as the seccomp messages are parsed, the not yet tracked syscall will be appended to the list of necessary syscalls. Another valid option at that point is a crash due to a bug. For return values other than bad syscall analysis is skipped. Before restarting AFL++ the Audisp Plugin is terminated to minimize performance overhead. This cycle is repeated until it is detected that fuzzing was interrupted by a timeout recognizable by an empty crash folder of AFL++, which leads to the final seccomp filter generation process.

Finally, we have to verify whether a syscall originates from the wrapper or the SUT itself. Therefore, we once again execute the binary in standalone mode whenever a new syscall has been found. In this way, we can check if the syscall originated in the wrapper or the binary and a syscall bitmask delta is created accordingly. This delta is subtracted in the last step, “Final Seccomp Filter Generation”.

Final Seccomp Filter Generation: In this final step, syscalls that were only necessary for the Seccomp-Wrapper, like shared memory operations for AFL++, have to be removed. Since this delta bit vector was created during fuzzing, it is just removed from the complete list of syscalls.

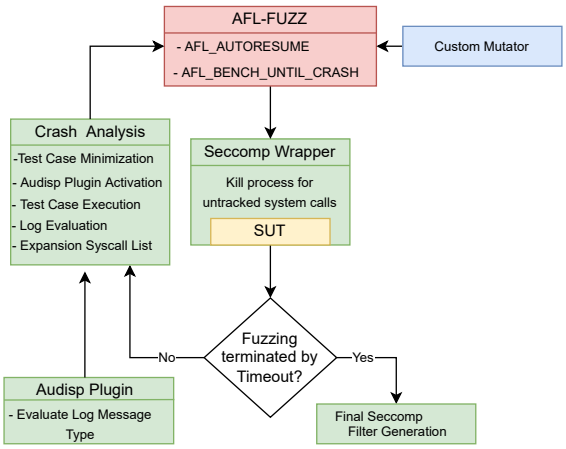

Figure 2: Fuzzing Process in Detail.

4 DESIGN TRADE-OFFS

When designing our tool and the crash analysis cycle above, we had to make different trade-offs that we explain in this subsection. Some of them may be overcome in future work.

The `BENCH_UNTIL_CRASH` cycle was introduced because we experienced instabilities with Audisp plugins. We tried different approaches like monitoring every syscall with syscall audit filters instead of using seccomp filter, but the queue to such a plugin was not capable of handling the number of events. This is why we decided to analyze after each crash via `BENCH_UNTIL_CRASH`. If it is possible to overcome these issues, it is not necessary to fuzz in `BENCH_UNTIL_CRASH` mode.

The invocation of `afl-tmin` to minimize test cases after each crash in the analysis phase is not necessary for the program behaviour. Furthermore, this adds additional executions of the SUT and therefore lowers the performance. However, we relied on it to debug, and manually analyze the resulting filters. It may be useful to add an option to disable this behavior.

The seccomp pre-evaluation step just resembles a trade-off between increased initial time overhead and reduced fuzzing time, which also could be skipped.

The system assumes that every bad syscall return value is indicative of a new syscall being found, but neglects the problem of nested seccomp filters. However, some programs, like `file`, also use seccomp sandboxes and therefore rely on seccomp filters. In that case, fuzzing may stop due to crashes caused by the nested seccomp filter. This situation is detected during the analysis phase, but it still lowers the performance due to unnecessary executions.

5 EVALUATION

In the evaluation section we compare the syscalls that our fuzzing based tool find with the list of syscalls we discovered by manually testing different program behaviours and recording them with `strace`. Binaries, which are suitable test candidates, should rely on different sets of syscalls as their argument vector changes. However, this requirement is not easy to fulfill, since most of the coreutils rely on a consistent set throughout the execution of different paths with only small changes to them. The binaries `ls`, `file` and `diff` have varying syscalls depending on their set of CLI flags. Therefore, they make for a good benchmark to evaluate our tool on. The configured fuzzing time was three minutes in every case to be able to compare the different approaches. However, the real fuzzing time differs because the timeout is reset after each crash and AFL is restarted in these cases.

As shown in table 2, 3 and 4, `ls` from the coreutils, `diff` from diffutils and `file` are candidates, which satisfy this requirement and at the same time have a variety of different syscalls. The table of `diff` was shortened and all baseline syscalls produced without argument flags were removed to improve clarity. Furthermore, `file` is especially interesting, since it utilizes seccomp filters itself as default configuration and on the other hand relies on different combination of arguments to reach certain syscalls like `vfork`.

In our evaluation, the AFL specific instrumentation was responsible for the syscalls `shmat` and `shmctl` being added to the list of necessary syscalls, but were subtracted in the policy finalization step.

The fuzzing time, covered paths and identified CLI flags are listed in table 1. The result looks promising for `ls`. AFL reached 394 different paths in three minutes of fuzzing time and covered thereby all the manually identified syscalls (see table 2). Because `ls` displays the most trivial test case and does not call any corner case syscall like `execve` the according table is not shown here. The tested version of `ls` was 9.0.36. Our tool was also able to find all manually identified system calls for `file` (see table 3) in 60m. The tested version of `file` was 5.39.

For `diff` we observed less than 100% coverage, but still all except for two of the manually found syscalls were identified (see table 4). The syscalls `stat` and `lstat` were not found. `diff` has many CLI flags, and a three minute timeout might not be enough to find the necessary combination to trigger the syscall. On the other hand, it is also possible that `stat` was never seen because `diff` is never executed without arguments during fuzzing. The tested version of `diff` was 3.7.

Table 1: Overview Evaluation Statistics.

SUT	Covered Paths	Fuzzing Time	Found CLI Flags
ls	394	3min	111
file	157	60min	63
diff	490	4min 23s	177

Table 2: Evaluation with ls.

required*
read, write, close, lseek, mmap, mprotect, munmap, brk, rt_sigaction, rt_sigprocmask, ioctl, pread64, access, socket, connect, uname, getcwd, readlink, capget, statfs, prctl, arch_prctl, getxattr, lgetxattr, futex, getdents64,set_tid_address, exit_group, openat, newfstatat, set_robust_list, prlimit64, statx
not found
-

6 DISCUSSION

The good detection rate for `ls` is not surprising due to the independence of argument vectors and the missing necessity of providing a valid file path. It can also be executed without any CLI flags and it will still return successfully. In contrast, `diff` and `file` rely on a more complex syntax, which expects two files in most cases for `diff` or argument dependencies in case of `file`.

Although our tool found 100% of the manually identified syscalls in `file`, it poses a unique challenge for our tool. This is because fuzzing aborts any time `-p`, `--preserve-date`, `-z`, `-Z`, `--uncompress`, or `--uncompress-noreport` is used. The process will crash because of nested seccomp filters, even though all used syscalls already have been tracked and no new insights are generated. This is the reason for the comparatively long fuzzing time this case.

In general, AFL++ only uses one core per fuzzing instance. In our tool, the performance could be increased significantly by implementing parallelized fuzzing. Furthermore, the problems of nested seccomp filters lead to a significant impact on performance. But overall, the quality of the generated seccomp filters can be described as good, because they are able to cover all high frequency paths and their needed syscalls as well as nearly all low frequency paths.

Table 3: Evaluation with file.

required*
read, write, close, stat, fstat, lstat, lseek, mmap, mprotect, munmap, brk, ioctl, pread64, access, mremap, prctl, arch_prctl, getdents64, exit_group, openat, newfstatat, utimensat, seccomp, pipe, rt_sigaction, vfork, execve, wait4, dup2
not found
-

Table 4: Evaluation with diffutils.

required*
access, arch_prctl, brk, clone, close, dup2, execve, exit_group, fnctl, ioctl, lseek, mmap, mprotect, munmap, newfstatat, openat, pipe, pread64, read, rt_sigaction, sigaltstack, wait4, write, stat, lstat
not found
stat, lstat

7 CONCLUSION AND FUTURE WORK

In this paper we presented a way to find syscalls in a SUT using fuzzing techniques. To this end, we utilized seccomp and auditd to monitor the program while executing it repeatedly with different inputs using AFL. We considered command line flags and fuzzed them separately from additional input to increase coverage using a custom mutator. Our tests with `ls` and `file` have shown that we find most syscalls in a reasonable amount of time, even if they are hidden behind different command line flags. However, some problems exist: If the SUT itself uses seccomp filters (e.g. `file`), the nested filter may cause crashes which are difficult to distinguish from newfound syscalls. Similarly, complex requirements, such as flags that require compressed files as input, cannot easily be treated with our approach.

In the future, working on these issues would be the next step. Combinatory command line flags can already be mitigated by a longer amount of testing. However a smarter setup for specific formats, such as the aforementioned compressed file flag, either requires manual work, or a more sophisticated approach to the fuzzer. Nested seccomp filters might be overcome by using other tracing methods. It might also be possible to resolve this issue using `SECCOMP_RET_ERRNO` to disallow the seccomp syscall while tricking the SUT into thinking that is has succeeded.

*The required syscalls were determined by a manual analysis.

Currently, our approach requires the source code to be available and to be compiled with AFL instrumentation. An experimental feature of AFL using QEMU can be used to instrument binaries. The symbolic execution in the beginning relies on the source code as well. It would be interesting to investigate, how well the analysis works in black-box settings, where symbolic execution does not run and a first set of CLI flags is instead determined using static analysis.

REFERENCES

Böhme, M., Pham, V.-T., and Roychoudhury, A. (2019). Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506.

Canella, C., Werner, M., Gruss, D., and Schwarz, M. (2021). Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, CCSW '21, page 139–151, New York, NY, USA. Association for Computing Machinery.

DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., and Kemerlis, V. P. (2020). Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Ghavamnia, S., Palit, T., Mishra, S., and Polychronakis, M. (2020). Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766.

Lopes, N., Martins, R., Correia, M. E., Serrano, S., and Nunes, F. (2020). Container hardening through automated seccomp profiling. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, WOC'20, page 31–36, New York, NY, USA. Association for Computing Machinery.

Rizzi, E. F., Elbaum, S., and Dwyer, M. B. (2016). On the techniques we create, the tools we build, and their misalignments: A study of klee. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 132–143, New York, NY, USA. Association for Computing Machinery.

Zeng, L., Xiao, Y., and Chen, H. (2015). Linux auditing: Overhead and adaptation. In *2015 IEEE International Conference on Communications (ICC)*, pages 7168–7173.