System Call Sandboxing

Comparing static and dynamic analysis and filter generation

Petr Khartskhaev
Supervisor: Alexios Voulimeneas
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Petr Khartskhaev
Final project course: CSE3000 Research Project
Thesis committee: Alexios Voulimeneas, Przemysław Pawełczak

Abstract

All complex programs are bound to contain software bugs, of which some could be exploited. These exploits rely on the application being able to start – or become – a process that it should not normally. To exploit these applications in this way, the attacker needs the operating system's kernel to escalate the attacked program's permissions or to start another process. Sandboxing is a way of stopping the attacker by limiting, which calls a program can make to the kernel – if a program never does start new processes, the sandbox application intercepts all requests (system calls) to do so. In this paper, I develop a script to collect a list of a program's used system calls (shortened to syscalls) obtained dynamically by an external tool, and compare its output to existing – static – solutions, which extract a program's system calls by examining its machine code from the binary. Three out of four tested programs functioned correctly after applying the filter, and the fourth one's failure may be caused by the program I used to perform the sandboxing – firejail. All four resulting lists were almost half the size of the next best performing tested application. Further research should be done into why one of the applications failed in the same scenario that was used to record the syscalls, and how each syscall's arguments can be filtered to further decrease the possible attack surface.

1 Introduction

In common modern operating systems, all interaction between normal, user-mode programs and the kernel – the core – is handled using system calls (syscalls) [1] (Section 2.3). Since the kernel has maximum privileges, which could be exploited by malicious actors, it is essential this communication is done safely with as little room for potential attacks as possible.

Sandboxing is a way to possibly prevent a program from invoking an unnecessary or malicious call in case of the program being compromised, by means of running it in an environment that limits what it can do [1] (Section 17.11.3) (e.g. a program that never calls `setuid` [14] can be blocked from invoking it in order to prevent exploits). There are two possible approaches to this problem: static and dynamic. In a static approach, a specific program is first analysed without being executed, then a profile is generated for which system calls to disallow [2], or optionally, the execution of the program is divided into different stages, with a different profile being applied to each (e.g. initialisation and main cycle in web servers) [5].

A dynamic solution is to run the program in multiple different scenarios, observe exactly which system calls are executed, and create a filter from the result. The latter approach obviously requires much more effort in the form of constructing the aforementioned scenarios, running them with a tracing tool such as **strace** [16], then performing the analysis. On the other hand, it provides a much more detailed insight into the specific parameters and stages of execution of the program, as opposed to the existing static solutions. This allows for creating of tighter filters, further preventing the misuse of programs.

The objectives of this paper are to:

1. test the dynamic approach on programs such as *ls* [15] and Openbox [10];
2. compare this approach to existing static techniques from previous work [2, 4, 5, 3, 6].

The most important finding should be how many system calls are missed by the static analysis tools and how many are included incorrectly. This can help in further studies researching whether and under which circumstances it is preferrable to employ dynamic analysis instead of static.

The paper has the following structure: Section 2 will provide background and related literature, as well as the methodology. Section 3 will describe the way in which my dynamic analysis tool for system calls works. The results of the comparison will be presented and explained in Section 4, and Section 5 will report about the ethics and reproducibility of this paper. Section 6 will compare the results to ones from previous papers and attempt to explain possible inconsistencies. Finally, Section 7 will include a conclusion and discuss possible future work.

2 Background and methodology

2.1 Static analysis

Static analysis can be done in two ways – scanning the source code, and scanning the binary of a program [2]. The main idea is to locate all references to system calls in the program and its own dynamically linked libraries, as well as all references in global library functions that the program calls (e.g. the C library).

The specific method of analysing a program's source code is described in Canella et al. and Ghavamnia et al. (Aug. 2020) [2, 5], while the methods of analysing the binary are discussed in Ghavamnia et al. (Oct. 2020), DeMarinis et al. and Petrich [4, 3, 6].

Canella et al. [2] propose a tool called **Chestnut**, which includes static analysis tools for both source code and binaries, as well as a dynamic component that refines the filter during runtime. The source code tool, **Sourcalyzer**, uses the LLVM compiler framework to find syscalls in source code. This approach is also used by Ghavamnia et al. (Aug. 2020) [5] in their temporal syscall analysis tool, which generates separate profiles for a program's execution phases.

Chestnut's binary analysis tool, **Binalyzer**, searches for *syscall* instructions and finds the syscall numbers for each of them using symbolic backward execution. This is also the method used by Ghavamnia et al. (Oct. 2020) [4] in their Confine tool and Petrich's **Callander** [6].

2.2 Dynamic analysis

Dynamic analysis involves running the program to retrieve its syscalls. DynBox [7] performs "partial-order analysis" on a program, which enables it to disallow syscalls that are guaranteed to not be used again at runtime.

In my method, separate scenarios are constructed for each program, in which (ideally) every possible function of the program is executed. All invoked syscalls are recorded using a tracer, and those are then compiled into a filter alongside their parameters. This should

enable the construction of a more specific and tight set of possible parameters for each syscall, as well being very easy to split into different stages of execution. A significant downside of this approach is that it requires much more effort, pertaining to both the need to construct a sufficient set of scenarios to run for each program, and to run the program in those scenarios. If the scenarios are not comprehensive enough, there is a risk of the resulting filter being too tight and not allowing the program to run as expected.

2.3 Applying the filter

The references can then be compiled into a list of allowed syscalls and their possible parameters (as well as the stage of execution of the program), and applied as a filter to the program in question. This can be done by modifying the source code or binary [2], or running the program in a wrapper with the filter applied (e.g. firejail, Callander, or Docker's seccomp profile option [9, 6, 12]).

3 Automated dynamic analysis

After constructing the scenarios and recording a trace of all invoked system calls, it must be analysed. The output should consist of a dictionary of syscalls as keys with the values being allowed values for each of the six parameters (the maximum amount of arguments for a syscall in x86-64 systems is 6 [17, 11]). This dictionary can then be converted into a filter for available sandboxes or wrappers.

To achieve this output, the runtime of the analysis tool must be divided into several steps:

1. parse the file produced by the tracing tool into separate syscalls with arguments;
2. combine the syscalls into a dictionary with a set of all used arguments;
3. parse the arguments into addresses, integers, strings and structures;
4. combine the arguments within each syscall to form a list of possible arguments.

It is assumed that the tracing tool produces an output where each line contains one syscall in the format of:

`<time> <call>(<comma-separated>, <arguments>) = <return value> <duration>`

as this is the format produced by running `strace -tT -o <output file> <command>`.

This file can then be manually separated into execution phases (either using the timestamps or comparing with another file which the result of only running one of the phases) and passed into the script, which extracts the syscall name, arguments and return value from each line. The arguments for each are then parsed and separated into an array.

Afterwards, this data is unified into a map from syscall to an array of arguments (length six), where each element is a set of recorded arguments. Then, different algorithms can be used to consolidate different types of argument (e.g. addresses should not be unified since

they will be different on each run, differently sized structures can have some values that are always the same, and some parts which are always different, etc.). I opted for a very simple one.

The syscall-to-arguments map is iterated through. If a syscall's argument's trace only consists of memory addresses (e.g. `brk`'s only, 0th argument is always a memory address, be that `NULL` or any other), the set is generalised to "address" as a program's memory space changes every time it is run. Other sets of argument values, which are larger than 3, are generalised to "any", and all remaining sets are left in their original state (e.g. if `write` is only used with two strings of lengths 1 and 13, the third set in `write`'s argument array will be the set $\{1, 13\}$ and strings of no other lengths will be able to be written).

All this results in a valid whitelist of syscalls along with possible arguments, which can be imported into seccomp, or passed into a sandbox program such as **firejail** [9].

Because of firejail's implementation, only the syscall whitelist is used (without argument limitations). For an explanation see Subsubsection 6.1.2. Since the first syscall in any strace output file is always `execve` [8], the script ignores the first line of every file passed to it. This approach works with firejail, but with other popular seccomp sandbox applications it might not.

4 Experimental Setup and Results

4.1 Experimental Setup

The programs to be tested are as following:

1. A simple "Hello World!" program written in x86 Assembly without the use of the C library
2. A simple "Hello World!" program written in C
3. The *ls*[15] utility from the GNU Coreutils
4. Openbox[10], a window manager for Linux systems

The reasoning behind the "Hello World!" programs was to observe how the C library alters the run of a program. The *ls* utility was selected because it has many possible scenarios, i.e. the large amount of flags and possible directory structures to display. Openbox was chosen due to its versatility such as launching and closing applications; moving, resizing, minimising and maximising windows; picking options from the menu etc.

Constructing scenarios for the first two programs is trivial, as there is only one. For *ls*, which has 60 different flags, each one must be tested as well as running the program in different environments (i.e. an empty directory, a directory with files and subdirectories, a linked directory, a directory with no read permissions). For *ls*, I ran it in the aforementioned environments for different flags – no flags, all items (`-a`), almost all items (`-A`), long listing (`-l`), author (`--author`), human readable (`-h`), inode (`-i`), reverse order (`-r`), recursive (`-R`), unsorted (`-U`), and allowed combinations thereof.

Openbox has a variety of features, therefore a scenario whereas many of them are used must be constructed, along with a scenario where the program is exited immediately after launch in order to isolate the initialisation phase. For the long scenario, I opted for the following steps:

- Edit the configuration to launch the terminal with the Alt+Q keyboard shortcut
- Launch Openbox in a Xephyr [18] window
- Launch the terminal via the shortcut
- Re-size the terminal
- Switch to another desktop via a middle click
- Open the menu
- Select Firefox
- Minimise Firefox via the title bar button
- Switch to the original desktop
- Close the terminal via the title bar button
- Launch the terminal
- Close the terminal using the `exit` command
- Open the menu
- Select *System -> Reconfigure Openbox*
- Open the menu
- Log out

After running each scenario with *strace* (the command being `strace -Tt -o <output file> <program>`, where the `-Tt` flags indicate that a global timestamp and duration information will be added to each syscall), the parsing and analysis script is run to extract the list of syscalls and their arguments. That list is then:

- used in a *firejail* [9] environment running the program in question (also with every scenario, to test if the filter is not too tight);
- compared to the list of allowed syscalls for the same program produced by Callander and a modified version of Chestnut's Binalyzer

4.2 Results

Running with firejail

Having recieved a list of allowed syscalls from my script, I used firejail with the following command:

```
firejail --noprofile --seccomp.keep=<list of syscalls separated with commas> <binary>
```

The first three applications worked flawlessly (i.e. every scenario ran the same and with the same outputs), but it was only possible to interact with already opened applications and not launch new ones. Launching a new application would result in a window with the error: "Failed to duplicate file descriptor for child process (Operation not permitted)".

Comparing to existing tools

I summarised the comparison of the number of allowed syscalls from each tested solution in Table 1. One can see that the dynamic solution, where syscalls are extracted from a trace, always allows fewer syscalls than programs employing the static analysis approach.

I also examined how the output of the static tools differed from mine – how many syscalls did the tool output that mine did not (labelled *Extra*), and how many syscalls did my tool include that the other did not (labelled *Missing*). I then ran each program with Callander and with the same firejail command using filters recieved from Binalyzer and noted in the table (rows labelled *Works*) whether they produced the same, expected output. In one case the program failed to launch and in two cases it did not function properly. The possible reasons behind this are discussed in Subsubsection 6.1.3.

For the Assembly "Hello World!", where both its system calls are written explicitly in the source code, Callander locates three more – `gettimeofday`, `clock_gettime` and `clock_getres`. Chestnut finds circa 3.5 times more syscalls in *ls* than Callander, and 6.8 times more than are really used.

Table 1: Allowed syscalls for each program found by each tested solution

Tool		"Hello World!" (Assembly)	"Hello World!" (C)	*ls*	Openbox
My Solution	Total	2	18	39	59
	Works	Yes	Yes	Yes	No[2]
Callander	Total	5	42	77	149
	Extra	3	30	46	94
	Missing	0	6	8	4
	Works	Yes	Yes	Yes	No[3]
Chestnut[1]	Total	2	154	266	287
	Extra	0	138	234	233
	Missing	0	2	7	5
	Works	Yes	Yes	No[4]	No

5 Responsible Research

All tools to reproduce this research have been provided on GitHub: `https://github.com/felacek/dynamic-syscall-filtering`. That includes the output of all three tested tools – Callander in Docker with Ubuntu 22-04 (the image can be found on the Callander GitHub page `https://github.com/rpetrich/callander`), Chestnut on the local system, Arch Linux with kernel `6.9.5-zen1-1-zen` and glibc version `2.39+r52+gf8e4623421-1` using Python 3.8, and my script on the same system using Python 3.12.

[1] The code was modified, see Subsubsection 6.1.1
[2] Generally works, but cannot launch applications
[3] Launches, but works cannot interact with applications without errors
[4] Lists all files and directories but no properties

As with all cybersecurity tools, this research could come in useful to malware developers and other bad actors, but seeing as this category of tools already exists and this paper is concerned with the preferred method of compiling syscall whitelists, this should not give bad actors any advantage.

6 Discussion

Overall, the sets of syscalls obtained were generally much smaller than the ones produced by static analysis tools. I did not compare the argument filters produced by my solution and Callander, mainly because I could not test whether mine were or were not too tight, but also because Callander returns argument filters based on its analysis of dynamic libraries like glibc, ignoring what the program itself provides.

6.1 Difficulties

A difficulty that I encountered during testing was using the static analysis programs since I was not able to install any of them without making alterations to the code. Out of the six previously existing tools, I was able to run two, those being Chestnut (Binalyzer only) and Callander.

6.1.1 Modification of Binalyzer

As mentioned previously, I had to modify Binalyzer's code for it to work and likely introduced a bug, because the results presented for *ls* by Canella et al. [2] show that Binalyzer detected 39 syscalls, not 266. I was unable to fix this bug while keeping the rest of the code working.

6.1.2 Other methods of enabling seccomp profiles

Another difficulty surfaced when attempting to run Docker containers with seccomp profiles, since Docker required a list of specific enabled syscalls that were not always necessary to create the container itself, with no option to enable the hardened profile after container creation.

I also attempted to use a **seccomp-filtered-run** [13] to run programs with custom seccomp profiles without modifying the binary, but found that it enabled the profile too early, before forking and executing the program itself, which resulted in an error. Firejail was therefore the only available application I used to run programs with custom profiles easily, with the drawbacks of:

- not allowing one to specify filters for arguments;
- not working with the way Openbox interacts with the X server to launch applications. More research should be done to find out why it behaves this way, when the (rootless) X server is running with no restrictions.

6.1.3 Running with static analysis tool outputs

Another problem surfaced when attempting to run Chestnut's output list with firejail, as *ls* had no permission to read any properties of the files it listed and Openbox failed to launch.

As both the analysis and the sandbox were run on the same machine with the same system, possible incompatibilities should be ruled out. Further research must be done to understand why this happened.

7 Conclusions and Future Work

In this paper, I demonstrated a way to extract a program's used system calls after running and tracing it, and compile them into a filter to be passed into a sandboxing application. I also compared it to two pre-existing solutions with a different approach, namely one which analyses a given program's binary statically. I tested my approach on 3 simple programs (two "Hello World" binaries and *ls*) and 1 complex one (Openbox), ran them and compared the resulting filters to ones output from Chestnut, a binary static analysis application.

I used firejail as the agent imposing syscall filters onto the programs, and all tested programs aside from Openbox worked perfectly with the list of allowed syscalls provided to them. The script to extract syscalls is fast and outputs the list of used syscalls along with generalised possible arguments for each one, as well as the amount of allowed syscalls and a firejail command which can be copied and run with the desired binary.

In comparison to static methods, the dynamic method described in this paper is more arduous to set up since one must make a scenario which uses every possible syscall in the program's code, but produces much tighter restrictions with (in most cases) all functionality in-tact. For the tested programs which ended up working with this approach, it allowed circa 44.5% fewer syscalls than the next best tested tool (Callander).

This shows that dynamic syscall analysis performs well for simple and more complex programs, although this conclusion is hindered by the fact that it was difficult to run the existing static tools and they might perform better under specific conditions. Very large and complex programs have not been tested and the fact that comprehensive scenarios could be much harder to construct for that type of application means that dynamic syscall analysis is probably not the best method for applications.

For future work, it is imperative that a new sandboxing application be found or developed, which can provide more user control and fewer quirks. It should be understood why firejail prevented Openbox from working properly. Argument filters per syscall should also be tested and compared to other analysis applications' results.

References

[1] Avi Silbershatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* Tenth Edition. John Wiley & Sons, Inc., 2018.

[2] Claudio Canella et al. *Automating Seccomp Filter Generation for Linux Applications.* 2020. arXiv: `2012.02554 [cs.CR]`.

[3] Nicholas DeMarinis et al. "sysfilter: Automated System Call Filtering for Commodity Software". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 459–474. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/demarinis.

[4] Seyedhamed Ghavamnia et al. "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 443–458. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/ghavanmnia.

[5] Seyedhamed Ghavamnia et al. "Temporal System Call Specialization for Attack Surface Reduction". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1749–1766. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia.

[6] Ryan Petrich. "Linux Sandbending: Binding Program Behaviors without Binding Ourselves". In: Presented at All Day DevOps 2023, 2023. URL: https://github.com/rpetrich/callander.

[7] Quan Zhang et al. "Building Dynamic System Call Sandbox with Partial Order Analysis". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622842. URL: https://doi.org/10.1145/3622842.

[8] *execve*. URL: https://man7.org/linux/man-pages/man2/execve.2.html.

[9] *Firejail*. URL: https://firejail.wordpress.com/.

[10] *Openbox*. URL: http://openbox.org/wiki/Main_Page.

[11] *seccomp*. URL: https://man7.org/linux/man-pages/man2/seccomp.2.html.

[12] *Seccomp security profiles for Docker*. URL: https://docs.docker.com/engine/security/seccomp/.

[13] *seccomp-filtered-run*. URL: https://gitlab.com/patlefort/seccomp-filtered-run.

[14] *setuid*. URL: https://man7.org/linux/man-pages/man2/setuid.2.html.

[15] Richard M. Stallman and David MacKenzie. *ls*. URL: https://man7.org/linux/man-pages/man1/ls.1.html.

[16] *strace*. URL: https://man7.org/linux/man-pages/man1/strace.1.html.

[17] *syscall*. URL: https://man7.org/linux/man-pages/man2/syscall.2.html.

[18] *Xephyr*. URL: https://freedesktop.org/wiki/Software/Xephyr/.