Investigation of the Over Approximation in System Call Set Extraction

Brian C. Tracy
Brown University CS2951U — 2021

Abstract

System calls (syscalls) give user space programs the power to do privileged operations like accessing the network, modifying the filesystem, and creating new processes. Likewise, syscalls also give user space programs the power to do dangerous things (flood the network, delete the filesystem, kill other processes). When compromised, everything that a user space process can do with syscalls becomes a tool for the attacker to exploit. To defend against the creation of powerful "confused deputies" when a user space process is compromised, some systems make only the smallest required subset of the entire syscall interface available to processes, in alignment with the principle of least privilege. `sysfilter` is such a defense mechanism that works by statically analyzing programs to determine which syscalls they need, and which they do not. Determining this subset of syscalls is non trivial, and because false positives (the blocking of a syscall that is legitimate) are unacceptable, the set of allowed syscalls must be over approximated. This paper investigates the extent to which `sysfilter` over approximates the allowable syscall set by comparing the static analysis predictions from several test programs to their actual syscall usage, as determined by dynamic analysis.

1 Introduction

Historically, the entire system call interface has been available to every user space process. This allows any process, regardless of how narrow its goals may be or what it requires from the operating system, to perform a large variety of dangerous actions. As an example, there is generally nothing stopping a compromised `cat` program from opening up a network socket or deleting a file, despite these being things that would never legitimately be needed by such a program.

This rather glaring violation of the principle of least privilege has been known about (probably, like all things, since the '70s) for a long time and has been addressed in the Linux kernel for at least 15 years [3].

Beginning in 2005, programmers could use the `seccomp` facility in the kernel to prevent their processes from performing unnecessary system calls through manual blacklisting. This is a great step forward for modern programs, but leaves legacy binaries in their original state of relative insecurity.

In an attempt to automate the process of determining which syscalls a program really needs, and which it can do without, `sysfilter` was created in 2020 [2]. This paper investigates the properties of `sysfilter` with respect to how good of an approximation it is able to make about the required set of system calls of a given binary.

2 How `sysfilter` Works

To harden a program with `sysfilter`, two steps are required. The first phase of the hardening, known as the "extraction" phase, determines which system calls a given program needs. The second phase, known as "enforcement", actually modifies the program itself to prevent it from making any sycalls other than those identified in the previous phase.

The extraction phase begins by identifying all potential system call sites by disassembling the target binary and looking for `syscall` instructions. From here, the value of the `%rax` register can be tracked to determine which exact syscall is being made at any given call site (in the Linux x64 ABI, `%rax` indicates the syscall number).

Once the set of all *potential* system calls sites are known, `sysfilter` gets to work on pruning it. To understand why the pruning is necessary, consider the fact that most programs dynamically link with the C standard library, which contains bindings for most syscalls. If the entire set was not pruned, `sysfilter` would just report that all syscalls (whose call sites are brought in by the C standard library) are used in every program.

To prune the set of all potential syscalls, something akin to dead code elimination is performed and any call sites that are known to be unreachable are removed. This process **is not completely solvable** and `sysfilter` will not be able to always remove *all* unused system calls.

Due to this fact, the final set of syscalls that the extraction phase returns will always be at least as large as the theoretical

set of syscalls that the program actually makes. The extent to which `sysfilter` over approximates this set is the focus of the rest of the investigation.

3 Tools

To assess how much larger the statically determined set of syscalls is than the set of calls the program will make during runtime, the program needs to be monitored while it executes. The `strace` utility performs this monitoring and will compile a list of all system calls (and other statistics) that a given program makes.

In addition, a simple python script is used to map between syscall names and numbers (as defined by Linux header files) for ease of understanding.

4 Code Coverage and Lower Bounds

When performing the dynamic analysis step (as instrumented by `strace`), it is important to understand *how much* of the program you are inspecting is actually being executing. To accomplish this, a code coverage tool can be employed.

For example, if dynamic analysis says that 10 syscalls were performed during the lifetime of a process, but the code coverage metrics say that only 50% of all code paths were taken, then we can not be confident in our figure. The un-executed paths could be responsible for making syscalls that we do not know about. Due to unknown potential of these code paths, our `strace` analysis will always be a lower bound on the number of syscalls a program could theoretically make, given any input.

To increase the confidence that the dynamically determined lower bound on the number of syscalls a program makes is actually close to the theoretical bound, we need to ensure that more of the program's code gets a chance to be instrumented. Code coverage is the metric that will lead to confidence in the generated lower bound, and thus the analysis works best on programs with high code coverage.

5 Degenerate Edge Cases

To better understand the precision of the tools being used in the investigation, several small test cases were created. Specifically, I wanted to know what the minimum set of syscalls both `sysfilter` and `strace` could extract/detect.

An obvious, yet illustrative, starting point is the standard C "Hello World!" program. `strace` indicates that 13 syscalls are made, most of which seem out of place for such a simple program. As suspected, `write`, `close`, and `exit` are present, but the rest are slightly less obvious.

These seemingly "extra" syscalls come from the program setup phase as performed by the C runtime. Before `main` gets called, and after it exits, lots of setup/teardown code needs to executed, and along with it, several system calls need to be made.

One unfortunate artifact of this process is that every program ends up calling `mprotect`, a rather dangerous system call that changes permissions of memory pages, and a common target for attack.

To get the absolute minimum set possibly detectable by `strace`, I created the smallest ELF I knew of and bypassed libc entirely.

```
GLOBAL _start
SECTION .text
_start:
    mov rax, 60 ; exit()
    mov rdi, 98
    syscall
```

For this tiny executable (which only directly calls `exit`), `strace` picks up calls to `mmap`, `mprotect`, `brk`, `access`, `execve`, `arch_prctl`, and `exit`.

This set will serve as the baseline for all `strace` measurements and will contribute to the dynamically determined lower bound of syscall usage for every program, even if they don't "really" use these syscalls (i.e: the programmer never intentionally calls them).

To determine the limits of the `sysfilter` tool, with respect to the smallest detectable set, the hello world program is again useful. `sysfilter` reports printing "Hello World!" involves 47 system calls, including some interesting choices such as `socket`, `connect`, and `kill`.

6 Pre Investigation Notes

Early results from the above tests seemed to indicate that for trivial programs, the bulk of `sysfilter`'s over approximation comes from non optimal dead code elimination / call graph pruning of shared libraries. The size of libc compared to these trivial programs means that we are not really testing how well `sysfilter` works on our code, but rather we have reduced the problem to how well `sysfilter` works on a virtually unused libc shared object.

For this reason, I introduce the notion of "pathological" and "standard" test cases. The former are programs whose authored source code is just a drop in the ocean of the shared libraries it imports, while the latter are more realistic programs that actually do use a wide array of system calls.

7 Investigation Process

Armed with `strace`, I set out to create and discover interesting test cases to probe the extent of `sysfilter`'s over approximation. The majority of the work was dedicated to the "standard" test cases and the less realistic "pathological" ones are included as bonuses. The overall process was as follows.

1. Identify or create an interesting test program
2. Isolate this test program and build it as a standalone binary for consumption by `sysfilter`
3. Execute as many code paths as possible in the test program, while it is being observed by `strace`
4. Determine the code coverage of the previous step to determine how complete of a picture the `strace` output gives
5. Sanity check the difference between the static and dynamic analyses and attempt to explain the result

8 Hello World++

Inspired by the surprising results of the hello world example, I decided to continue this line of reasoning to its most absurd extent. If shared libraries are causing them majority of unused syscalls, then I would create a program that does nothing, but requires as many shared libraries as possible to do it.

Hello World++ loads in 4 common shared libraries, calls 1 function from each of them, and then exits. One function from each is called to prevent the wholesale elimination of the respective shared library by the optimizer.

```
$ gcc hello-world++.c \
-lpthread \
-ldl -rdynamic \
-lreadline \
-lncurses
```

As a bonus, two extra shared libraries are brought in as dependencies of the intended 4.

```
$ ldd a.out
linux-vdso.so.1
libpthread.so.0
libdl.so.2
libreadline.so.7
libncurses.so.5
libtinfo.so.5
libc.so.6
/lib64/ld-linux-x86-64.so.2
```

`strace` indicates that this bulkier hello world issues 22 syscalls, while `sysfilter` predicts 97.

Notice that no code coverage metrics were gathered for the hello world programs. This is because a visual inspection is enough to verify that the source is branchless and that there are no hidden syscalls that could be invoked.

9 Real Programs

The pathological examples illustrated that `sysfilter` has a tough time handling largely unused libraries and that they are a major source of over approximation. However, my intuition was that the static analysis incurs a fixed "startup cost" for small programs that will be "absorbed" by larger programs that actually use the libraries they import. To test this hypothesis, I needed to perform the same analysis on a non trivial, preferably well known, program.

SQLite is an ideal benchmark because it can be amalgamated into a single C file, it has a built in test suite, and the developers have a fanatical passion for high test coverage [1]. The amalgamated `sqlite3.c` file is 230,000 lines long and contains the entire SQLite API. There are several ways to prepare this file for `sysfilter`.

The simplest option is to stick a dummy `int main(){return 12;}` method at the end of the file and hand it over to `gcc`. The resulting binary (which is linked with `-ldl -lpthread -lm`) now contains the entire SQLite3 API and a `main` function that are completely separated. This is the ideal situation for `sysfilter` to prune away all of the unused syscalls made by SQLite because not a single SQLite function is reachable from the main method. In fact, it appears that this is exactly what `sysfilter` is able to do!

`sysfilter` reports that the SQLite amalgam with a dummy `main` will use 63 syscalls, the exact same number (and list of calls) that `sysfilter` reports a blank program linked with `-ldl -lpthread -lm` will make. This is an amazing feat from `sysfilter`!

The next amalgamation is to not just have a dummy `main` method, but to provide a `main` that uses every core SQLite API. When this is given to `sysfilter`, a total of 73 syscalls are detected. This will serve as the smallest upper bound on the theoretical syscall set for SQLite.

10 Results

Running the SQLite tests while being instrumented by `strace` takes around 10 minutes for the *smallest* of the suites. During this time, 429471 tests are run and the original `sqlite3.c` file is 96.7% covered (as reported by `gcov`). A total of 55 different syscalls are reported. To interpret these results, we need to do some set arithmetic to determine which system calls are made by the test runner and which come from SQLite.

Assuming the soundness of `sysfilt` (it will never fail to report a system call that ends up actually being called), we can compute which system calls were issued solely from the test harness, and not the SQLite code.

Let the two sets $Trace = \{\text{strace output}\}$ and $Filt = \{\text{sysfilt output}\}$ be the sets of syscalls detected by `strace` and `sysfilt` respectively. To determine which were *only* present in the support code, let $Support = Trace - Filt$, where $|Support| = 17$ and is comprised of syscalls such as `utime`, `clock_nanosleep`, `symlink`, and other test related functionality. We know that these system calls *only* occurred in the

Table 1: Approximate Over Approximations by `sysfilter`

Program Name	`sysfilter`	`strace` (upper)
Hello World	47	13
Hello World++	97	23
SQLite (dummy main)	63	38
SQLite (rich main)	77	38

support code because any call *not* in the *Filt* set is definitely not being called by the SQLite code.

Knowing that 55 syscalls were made, and 17 of them were exclusively issued by the support code, we have an absolute **upper bound** on the number of syscalls made by SQLite of $55 - 17 = 38$. Contrasting this with the numbers generated by `sysfilter` gives us our over approximation.

11 Conclusion

`sysfilter` hardens a program by extracting, and then enforcing the set of system calls that the program legitimately needs. To ensure that needed syscalls are never excluded from this list, `sysfilter` must over approximate the necessary set. A large portion of this over approximation comes from the fact that programs load in huge amounts of external code from shared libraries, and guaranteeing that certain code paths are not taken is a non trivial task.

To quantify the actual extent of the over approximation in the results of `sysfilter`'s extraction phase, dynamic analysis can be performed to determine which syscalls are being made at runtime. This analysis shows that for small programs, the over approximation is quite large due to the disproportional size differences between application and library code. For larger programs that more fully utilize their shared libraries, the over approximation tends to be smaller.

References

[1] SQLite Open Source authors. SQLite Testing documentation. https://sqlite.org/testing.html.

[2] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, San Sebastian, October 2020. USENIX Association.

[3] Jake Edge. A library for seccomp filters. https://lwn.net/Articles/494252/.