Downright: A Framework and Toolchain For Privilege Handling

Remo Schweizer
Institute of Applied Information Technology
Zurich University of Applied Sciences
Winterthur, Switzerland
stephan.neuhaus@zhaw.ch

Stephan Neuhaus
Institute of Applied Information Technology
Zurich University of Applied Sciences
Winterthur, Switzerland
stephan.neuhaus@zhaw.ch

***Abstract*—We propose Downright, a novel framework based on Seccomp, Berkeley Packet Filter, and PTrace, that makes it possible to equip new and existing C applications with a request broker architecture. An extensive configuration language allows AppArmor-like configuration that supports programmers in building rules for system call parameter validation and result sanitization. Access to these privileged function calls can be restricted both within Linux kernel and user spaces.**

Downright's main strength compared to related approaches is that it implements a complete mediation request broker architecture, in which all system calls are vetted before execution, either by the kernel or by a request broker, which runs as another process. This isolates the main program from many failures due to programming bugs and attacks, which would have to pass not only the attacked program, but the request broker also. We argue that this makes acquiring and releasing elevated privileges easier and safer. Downright eliminates the need to write Seccomp programs, instead allowing policies to be expressed declaratively through a rich policy language.

We demonstrate the viability of this approach by instrumenting `nginx`, an industrial-strength web server and reverse proxy. While this instrumentation takes only a single line of code, we argue that even this effort can be avoided by suitable C runtime code. We show that Downright's overhead is substantial, halving `nginx`'s perfomance, but propose measures for optimisation.

***Index Terms*—security, privileges, request broker, seccomp**

I. Introduction

Let's say that you are writing an application, and you are worried what damage the application might do if taken over, e.g., through a remote code execution bug. Your attacker model is therefore either a local or remote attacker that can take over the application, either through a bug in your own code or a bug in thirs-party code, such as a library. Let us assume that you have already designed the application and expensive redesign is out of the question, e.g., because of external constraints such as frameworks that must be used for reasons outside of your control. We believe this to be a common scenario. How can you defend your application?

The attacker model is thus a local or remote attacker that can feed arbitrary input to your application, with the goal of escalating their privileges (for a local attacker) or to otherwise violate security assumptions, e.g., by overwriting system files, exhausting resources, crashing the system, executing attacker-controlled code, etc.

One way to defend yourself is by *privilege separation*, where one carefully separates the application into different independent parts that run in different processes, as different users, and communicate using well-defined interfaces. If you have already designed the application, however, and if the design does not aready feature such privilege separation, this will be hard if not impossible to retrofit.

Another way is to use *mandatory access control* to limit the side effects an application can have by limiting the types of system calls the application can successfully issue, and their arguments. For example, if after a certain point an application cannot open a socket, the application cannot exfiltrate data over the Internet, because in order to do that, the application would have to call *socket*(2)[1].

Existing tools for mandatory access control on Linux include AppArmor and SELinux, discussed later in this paper. In order to use them, policy files need to be installed and, in the case of SELinux, the machine rebooted for the newly installed policy to come into effect. This is doable, but needs support from the system administrator, which may or may not be available.

The latest tool to allow this type of mitigation is Secure Computation Mode with Berkeley Packet Filter (Seccomp-BPF), discussed in some detail later in this paper. Seccomp-BPF intercepts system calls. Its policies are written as short programs for an abstract machine, at the end of which there is a decision on what should happen with the system call (allow, fail with error, etc). Seccomp-BPF is a very promising solution because it adds both flexibility and speed to the checking process. But programs are hard to write manually and there is almost no tooling for Seccomp, making it hard to adopt. Still, some popular programs such as Chrome, Android, and Docker, use it [1], [2], [3].

Another form of privilege separation is the *request broker*. In a request broker architecture, two processes, the main application and the broker, work together. The main application has no rights to call, e.g., *open*(2), but the broker does. The main application would then forward the parameters for *open*(2)

[1] We write system calls like *this*(2), because system calls are traditionally described in Section 2 of the Unix manual and the manual page can be viewed by, e.g., `man 2 socket`. This also works with other sections of the Unix manual, e.g, *strace*(1) or *signal*(7).

to the broker, which would carefully check them and, if the checks were successful, call *open*(2) on behalf of the main application. The broker would then pass the resulting file descriptor back to the main application. The idea is that the broker is much smaller than the main application and therefore presumably easier to get right. The policy under which this broker would operate would not need to be installed by root.

Our framework Downright uses both Seccomp-BPF and a request broker architecture. It takes a program that has already been written, and a policy file. This policy file is compiled into a Seccomp-BPF program at compile time. When run, the program will automatically split into a main application and a request broker. The request broker will then install the precompiled Seccomp-BPF program and will further vet system call arguments and results. Figure 1 has a simplified developer's overview.[2]

Downright focuses on the individual application, which has security advantages and disadvantages. One advantage is that it also works when the application is run by people without administrator privileges and hence without privileges to administrate AppArmor or SELinux. A potential disadvantage is that the policy under which the application runs is fixed at compile time and is thus not available to administrators.

Downright therefore makes the following contributions:

- Downright complenets existing tools that aim at preventing security problems from occurring in the first place by limiting the damage that programs can do once they are taken over.
- Downright can equip C programs with a request broker, even when the application has already been written. No changes in the application's architecture are necessary.
- Downright works whenever the operating system supports Seccomp and PTrace, which includes practically all Unix-like operating systems. Downright can therefore be used when it is not clear whether the target machine has AppArmor, SELinux, or other mitigation mechanisms.
- Downright simplifies Seccomp-BPF tooling enormously, eliminating the need to write Seccomp programs and instead allowing policies to be expressed declaratively.

The remainder of this paper is structured as follows. After giving an overview of related work in Section II, we discuss in some detail the two pillars of Downright, Seccomp and PTrace, in Section III. We do this because we feel that an understanding of these two techniques is essential for understanding Downright's architecture, which we give in Section IV. We describe the experiments we did with Downright, concerning both usability and performance, in Section V and conclude and give an overview of future work in Section VI.

II. Related Work

Downright uses both system call monitoring (to contain an application's access to resources) and privilege separation (to limit damage in case an application is taken over). It uses Seccomp-BPF and ptrace for this, and these are described in more detail in Section III.

A. *Monitoring System Calls*

There is much related work on managing applications by monitoring and restricting their system calls, among the earliest are Systrace [4] and SubDomain [5]. Both follow a similar approach by confining the application's access to resources like system calls, files, or sockets. Both tools focus on specifically on *suspect* programs.

AppArmor [6] is a successor to SubDomain and hence also restricts program's access to resources, also using a per-program configuration file. Like SubDomain, AppArmor will only ever narrow, but not expand, the set of objects to which a program has access.

SELinux [7] is a project developed by the NSA originally for Mach and two research operating systems before it became integrated into Linux. It is a "Mandatory Access Control mechanism based on type enforcement". It is essentially a combination of traditional mechanisms, namely object labeling, role-based access control (RBAC), and, optionally, multi-level security. Security decisions are made based on assigning types to resources and then deciding based on RBAC whether access should be granted. SELinux has a reputation for being hard to configure and maintain.

B. *Privilege Separation*

Privilege separation [8] is a mechanism to prevent privilege escalation by splitting privileges across multiple independent execution entities communicating through an I/O channel like sockets. The goal, therefore, is to give each subprocess of an application minimal privileges to fulfill its intended task. Even if one of the processes gets hijacked due to bugs or insufficient input validation, damage to the entire system is limited.

An example using privilege separation is `OpenSSH`. This remote login tool has employed privilege separation since 2002 [9]. In early versions of `OpenSSH`, an attacker was able to craft messages that would lead to an escalation of privilege. This risk is significantly reduced in the new architecture: if the slave process is taken over by malicious input from the client, no harm is done, since the ultimate decision whether the client is successfully authenticated is made by the monitor, not the slave.

Another popular example of privilege separation is `qmail`. Compared to `OpenSSH`, the separation in this mail transfer agent is much more fine-grained: `qmail` consists of more than 24 separate Unix programs [10].

The advantage of such fine granularity is without question its security aspect. Small, independent, and mutually non-trusting modules drastically reduce the attack surface and also the difficulty for a developer to understand a single module. For example, in its now 23-year history, qmail has had zero acknowledged exploits [11]. We are phrasing this sentence carefully, acknowledging that there are some who claim to have found security flaws in parts of qmail, but there seems to

[2] This figure ignores many technical details that have nothing to do with Downright, such as linking with libraries; and also some that do, such as the possible use of PTrace for further manipulation of system call arguments and results, which are explained later.

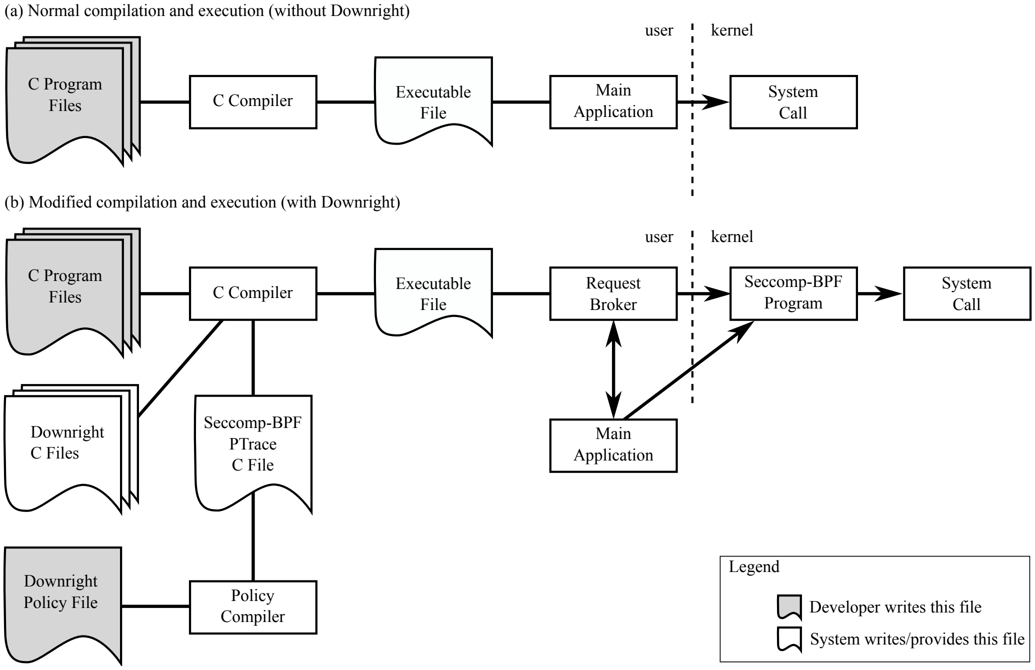

Fig. 1. Simplified developer's view of Downright. (a) Normally, one writes the program only and then lets system tools compile it into an executable. When that executable is run, the resulting process makes system calls. (b) With Downright, compilation includes a policy file. The resulting process is split into a request broker and a main application, whose system calls are additionally vetted with a Seccomp-BPF program inside the kernel. The C program files are the same as in (a); the only changes with respect to (a) are that a policy file needs to be written and that the build system needs to accommodate Downright, which in practice means changes to makefiles.

be a lack of proof-of-concept exploits that would violate any of qmail's guarantees in typical installations. A major contributor to this excellent track record is certainly qmail's design.

The concept of privilege separation therefore has many advantages. But controlling when and how system calls are executed is a good second line of defence. Qmail is again a case in point. In explaining what is and what is not covered by the qmail security guarantee, Daniel J. Bernstein writes [11]:

> In May 2005, Georgi Guninski claimed that some potential 64-bit portability problems allowed a "remote exploit in qmail-smtpd." This claim is denied. Nobody gives gigabytes of memory to each qmail-smtpd process, so there is no problem with qmail's assumption that allocated array lengths fit comfortably into 32 bits.

To not "give gigabytes of memory to [a] qmail-smtp process" can only mean that it should be best practice to limit the amount of memory that qmail-smtp can allocate with *brk*(2), and that is indeed the case: in many packaged versions of qmail, qmail-smtpd is started through a script that explicitly limits its available heap memory.

Also, limiting the execution of system calls is beneficial if, through some oversight, part of a privilege-separated process is taken over, even though that is less likely to happen than in a monolithic application. If the goal is to provide an easy-to-use framework to reduce the risk of privilege escalation for *any* Linux-based application, it is necessary to intercept system calls in order to mitigate the amount of damage that can be done if access to them is unrestricted.

Murray et al. present a concept where a process is disaggregated at the level of dynamically loaded libraries [12] by executing each library call in its own virtual machine. This protects its data from the access by potential malicious functions of other untrusted libraries. With this approach, privilege escalation attacks are still a serious problem because system calls are executed on the host system to which the main application has full access.

A completely different approach has been developed by Wang et al. with Arbiter [13]. The primary focus of Arbiter is to achieve fine-grained privilege separation in multithreaded applications. To accomplish this task, part of the memory management system of the kernel is rewritten to support permission bits. These bits can prevent mutual data access among the threads running in the same address space. Arbiter applies the concept of least privilege to shared data objects. However, this contradicts the objective of minimizing the privileges of the whole application. Portability also suffers due to the use of a custom Linux kernel.

Privman achieves privilege separation by running a privileged child process as a request broker [14]. Alternative versions for common used functions such as *priv_fopen*(3) for *fopen*(3) can then be used to outsource the execution to this specific request broker. Using this approach, an attacker gaining the privileges to execute arbitrary codes can just invoke *open*(2) directly in order to open a file. In this way, the security measures are circumvented.

Unlike Privman, in which every function call must be manually replaced by a Privman alternative, Privtrans extends the automation process [15]. With Privtrans, a developer can simply achieve privilege separation by adding annotations to the source code. The framework then automatically executes these privileged actions in a separate child process. Even though it offers more possibilities and a simpler interface than Privman does, it is prone to the same attack vector as Privman.

ProgramCutter takes a similar approach to Privtrans but enhances the concept in such a way that no expert knowledge about the software application is required [16]. The tool automatically detects and rewrites privileged actions at the system call level to use special wrapper functions provided by ProgramCutter. This approach assumes that the privileged sections that it generates are free from vulnerabilities and that the execution traces used to cut the application are complete. The last point is rarely the case, especially in large and complex software solutions.

SPL|T handles this problem and circumvents it by adding a configuration file and a statistical code analysis [17]. It provides a further solution to the problem when privileged parts call functions mutually. This particular problem is resolved by creating IPC-based communication gateways between the privileged parts of the application. Also in this approach, data validation and sanitization for the privileged functions must be implemented manually in order to prevent an attacker from misusing them through a loophole in the unprivileged process.

SOAAP is a recent attempt at automating at least some of the work that has to be done when separating privileges. The authors note that "application compartmentalisation remains an art rather than a science: identifying, implementing, and debugging partitioning strategies requires detailed expertise in both the application and security", which we can certainly see in action in the design and implementation of qmail; see above. They therefore leverage "semi-automated techniques, grounded in static analysis, dynamic analysis, and automated program transformation, to improve the developer experience".

C. Privilege Dropping

Unix[3] has another technique for running parts of a process with fewer privileges than other parts: privilege dropping, using real, effective, and saved user and group IDs; see, e.g., [18, Chapter 9].

While this sounds fairly straightforward, getting it right so that switching between the various user IDs does not lead to additional security problems is tricky [19].

One program that has traditionally used this approach for privilege control is sendmail [20]. Unlike qmail, sendmail is a monolithic program, which therefore relies on careful awareness of what privileges are required when, and therefore on when to drop root privileges and when to reacquire them. This is hard to get right and consequently, sendmail has been a never-ending source of security problems (admittedly not all related to privilege escalation due to improper privilege handling). Additionally, sendmail being a monolithic program means that if you compromise part of a sendmail process, you have compromised all of it, leading to potentially easier privilege escalation attacks.

Jenkins et al. present a different perspective on their approach to build a new concept for dropping privileges [21]. Adding security policies at the Application Binary Interface (ABI) level helps define semantic relationships between code and data. Like the approach of Murray et al., the concept solely focuses on restricting access to data. The access to system calls and its risk for privilege escalation attacks remain untouched.

D. Seccomp and BPF

Seccomp-BPF has no publicly available tooling support. From the fact that Firefox and other systems use Seccomp-BPF, it can be deduced that the creators of these systems must have some internal tooling, which is, however, not publicly available. There is tooling for BPF, in the form of BCC [22], but BCC seems not to be available for the reduced virtual machine that Seccomp-BPF uses.

III. Seccomp and PTrace

A. Seccomp and Seccomp-BPF

One recent attempt to facilitate managing privileges is Seccomp, introduced into Linux with version 2.6.12 in 2005 [23]. Once Seccomp is in effect, the only things the process can do is compute, *exit*(2), and *read*(2) from and *write*(2) to already open file descriptors.[4] This drastically limits a process's ability to do harm if taken over. For example, a traditional exploit with shellcode will not be able to execute a shell because *fork*(2) and *exec*(2) are not available. Memory similarly cannot be exhausted because *brk*(2) cannot be called. Attempts to make a system call that is not allowed lead to the process receiving a signal.

Obviously, this approach is highly secure, but also very restrictive. Some programs may want to avail themselves of most of the protections that Seccomp offers, but may have to run with third-party libraries that execute system calls not under the control of the original program. A more flexible architecture is given by Seccomp-BPF, in which Berkeley Packet Filter programs are written to make a decision whether a given system call should be allowed or not.

Seccomp-BPF programs are `int` arrays containing instructions for a virtual machine. This machine has an instruction counter and an accumulator, both of which can be manipulated through various instructions. Seccomp-BPF programs are assembled on the fly through C macros like `BPF_STMT` or `BPF_JMP`. There are opcodes like `BPF_LD` (for load), `BPF_JMP` (for conditional jumps) and `BPF_RET` (for returning a verdict on the system call). Jump targets are relative instruction numbers. Here is an example of a Seccomp-BPF program.

```
// Architecture check (see text)
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
```

[3] Comprising actual Unix as defined by whoever is holding the trademark at the moment, as well as the various BSDs, Linux, and lesser-known Unix-like operating systems out there.

[4] And return from a signal handler, but this should be rare.

```
        offsetof(struct seccomp_data, arch)),
BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K,
        AUDIT_ARCH_X86_64, 0, 7),

// exit_group(2) is allowed
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
         offsetof(struct seccomp_data, nr)),
BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K,
        __NR_exit_group, 6, 0),

// exit(2) is allowed
BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K,
        __NR_exit, 5, 0),

// setrlimit(2) fails with EPERM changing CPU limit
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
         offsetof(struct seccomp_data, nr)),
BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K,
        __NR_setrlimit, 0, 2),
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
         offsetof(struct seccomp_data, args[0])),
BPF_JMP(BPF_JMP | BPF_JEQ | BPF_K,
        RLIMIT_CPU, 2, 1),

// Action executions according to the rule
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO
         | (EPERM & SECCOMP_RET_DATA)),
```

The virtual machine on which this program runs has access to a C struct called `seccomp_data`. The various fields in that struct can be accessed with `offsetof`. For example, to access the member `arch`, we write `offsetof(struct seccomp_data, arch)`.

The first two instructions above check the kernel architecture. The instruction `BPF_LD` loads a value into the BPF machine's accumulator, the machine architecture in this case. The second line jumps (opcode `BPF_JMP`) if this value is equal to (opcode flag `BPF_JEQ`) the constant (opcode flag `BPF_K`) `AUDIT_ARCH_X86_64`. It jumps to the next instruction if they were equal (jump target `0`) and $7 + 1 = 8$ instructions ahead if not (jump target `7`).

Counting 8 instructions ahead gives a return statement (opcode `BPF_RET`) that returns the constant (opcode flag `BPF_K`) `SECCOMP_RET_KILL`. This causes the process to be sent a signal.

This highlights most of the commonly used features of Seccomp-BPF, but the block of statements commented "setrlimit(2)" presents a more complicated case. Here, we want to make *setrlimit*(2) fail with `EPERM`, but only if the process attempts to change the CPU time limit. First, we check whether *setrlimit*(2) is being called. If it is, we then also check the first argument to that syscall. We assume that the arguments to the syscall are passed in an **`int`** array `args`. If the argument indicates that the `RLIMIT_CPU` limit is to be changed, we jump to the last instruction in this program, otherwise, we jump to the instruction that returns `SECCOMP_RET_ALLOW`. The last instruction says to fail the system call without executing it (return value `SECCOMP_RET_ERRNO`), but not by killing the process, but with `errno` set to "permission denied" (`EPERM`).

It is obvious that Seccomp-BPF can provide much finer-grained control over system call execution than simply allowing or denying certain system calls. However, Seccomp-BPF programs have limitations compared to (non-Seccomp-)BPF programs:

- There is only one register. Hence, we cannot load the system call number and the arguments into separate registers to check them there.
- Jump offsets must be nonnegative. Therefore, loops are not supported and Seccomp-BPF programs will always terminate.
- It is not possible to inspect data behind a pointer. Seccomp allows checking the pointer itself, but not the data to which it points.
- Seccomp-BPF programs are hard to write.[5] While there is tooling for BPF generally, for example through BCC [22], this tooling is not available for the BPF machine that is employed by Seccomp.

B. *PTrace*

Seccomp can be made to interact with PTrace. PTrace is a very old technique, having been introduced in 1975 with Unix V6. It allows a process tracer to observe and control the execution of another process, called the tracee. Typically, PTrace is used to implement debuggers or application analysis tools that inspect not only system calls but also the overall internal state of the application and its registers. The most popular example of a PTrace application is *strace*(1).

In addition to the Seccomp-BPF return values `SECCOMP_RET_ALLOW`, `SECCOMP_RET_KILL`, and `SECCOMP_RET_ERRNO`, Seccomp also supports the return value `SECCOMP_RET_TRACE`, which allows a tracer to be notified right before the execution of a system call. It gives the tracer the possibility to perform the following actions:

- Inspect or modify the arguments of a system call.
- inspect or modify the return value of a system call.
- Resume the execution of a system call.
- Prevent the execution of a system call.
- Change the system call into another one.
- Terminate the tracee.

PTrace deals with multithreaded applications by pausing all threads until the tracer has finished its job. This ensures that no data inconsistencies can occur while the application is inspected. This will obviously degrade performance.

C. *Why Seccomp-BPF and PTrace?*

While it is clear that judicious use of Seccomp-BPF and PTrace can defend an applicaiton against the attackers modeled in Section I, it is also clear that these measures are very low-level. Why use something so restricted and low level? Why not write these policies in C, or BCC?

One reason for not writing the policies directly in C is that such policies must be installed at the operating system level, and having user-defined code run with operating system

[5] In fact, Reviewer Three caught errors in the jump offsets in the above example, which came about because the example was part of a larger Seccomp-BPF program that was then shortened for this paper, and the jump targets manually adjusted.

privileges is generally considered a bad idea. Remember that Downright's aim is not to stop, e.g., a buffer overflow from occurring, but rather to mitigate damage if it occurs. Thus, if such policies would run only in user-land, they simply could not defend against application take-over: the bugs would still be there and if the bug is in third-party code, such as a library, user-level code could not mitigate it.

We can't use BCC because BCC does not apply to Seccomp-BPF, since the virtual machine is different.

The other reason is that processes need to make system calls in order to have side effects: a process cannot exfiltrate secrets, for example, if it cannot issue *write*(2), for example. A process cannot allocate memory from the operating system if it cannot call *brk*(2). And so on. Restricting a process's access to system calls is therefore a very effective way of stopping the type of attack outlined in Section I.

IV. Architecture

A. PTrace and Seccomp

Figure 2 gives a detailed overview of system call execution with Downright. The process begins when the tracee attempts a system call. This invokes the kernel, which happens differently on different system architectures. On 32-bit based systems, interrupt `0x80` is triggered, but on 64-bit systems, the machine instruction `syscall` is used. In both cases, the Seccomp-BPF filter program will be executed. There are five possible outcomes:

- If `SECCOMP_RET_KILL` is returned, the application is terminated immediately.
- If `SECCOMP_RET_TRAP` is returned, the system call is not execued and a `SIGSYS` signal is sent to the application, which is caught by a Downright signal handler. In the current implementation, the handler is used for debugging purposes only. It allows a system call number to be printed, showing what system call caused the application to fail.
- If `SECCOMP_RET_ERRNO` is returned, the system call is not executed and the variable `errno` is set to `ENOSYS`, indicating that the system call failed due to a missing implementation. This error code is also used by PTrace if a system call is skipped.[6]
- If `SECCOMP_RET_TRACE` is returned, the system call is not executed and the tracer is invoked.
- If `SECCOMP_RET_ALLOW` is returned, the kernel will execute the system call normally.

The tracer waits for an invocation due to `SECCOMP_RET_TRACE`. The tracer's multiplexer calls a wrapper function based on the system call number. The wrapper function first reads the argument data from the registers and memory space of the tracee. Afterwards, the system call arguments can be examined and, if necessary, modified. Three different outcomes can occur:

- `Terminate` instructs the tracer to immediately stop the execution of the tracee.
- `Skip` sets the register `orig_rax` to `-1`, signaling that the system call should be skipped, i.e., no system call should be executed.
- `Allow`, on the other hand, keeps `orig_rax` unchanged.

In case of `Allow` and `Skip`, the tracer returns from its invocation and the execution path continues within the kernel. If `orig_rax` is modified to `-1`, the system call won't be executed, which is recognized as failed execution within the main application. When the `orig_rax` register is unchanged, the system call will be executed.

After the execution of the system call, the tracer is invoked yet again, and the same procedure is executed. The difference now is that the return data from the executed system call can now be checked and modified. For instance, this allows the data received by the system call *recvmsg*(2) to be read and manipulated before it is returned to the tracee. It would therefore be possible to change the data that an application reads from a file.

Contrary to the first invocation, only two return values are supported at this stage, `Terminate` (terminating the tracee) and `Allow` (returning to the kernel and then back to the tracee). After that, the tracee continues to run until the next attempt to execute a system call is made.

Downright also supports a debug mode, which is sometimes needed to understand which system call led to the termination of an application. If the `debug` flag is set in the configuration file, Seccomp actions will be emulated within the tracer. The tracer will then print debug information and emulate the action that Seccomp would have performed.

B. Configuration Language

The configuration of the framework consists of two files. One is a C-based source file defining the system call prototypes together with some annotations to specify the special behavior of some arguments. This file will in general have to be written only once, and will need modification only when new system calls need to be supported. The other is an INI-type file containing the rules for the system calls. This file is the main configuration file for the application and will have to be written from scratch for each application.

1) System Call Prototypes: The C-based configuration file with the default name *sec_syscalls_conf.c* needs to be written only once and needs to be amended only when new system calls come along or when the semantics of existing system calls change. It contains the following information:

- system call arguments with their type,
- dependencies among single arguments,
- group names for arguments,
- headers used by a system call,
- implementation of the system call check routine,
- custom types and function definitions.

[6] Choosing an appropriate value for `errno` is tricky, because we do not want to repurpose an error code that is already in use to signal other failures. We think that `ENOSYS` should be sufficiently rare to reliably signal to the application that the system call was forbidden by Seccomp. `ENOSYS` should be rare because normal programs go through the standard C library, which should have stub routines only for those system calls that actually exist.

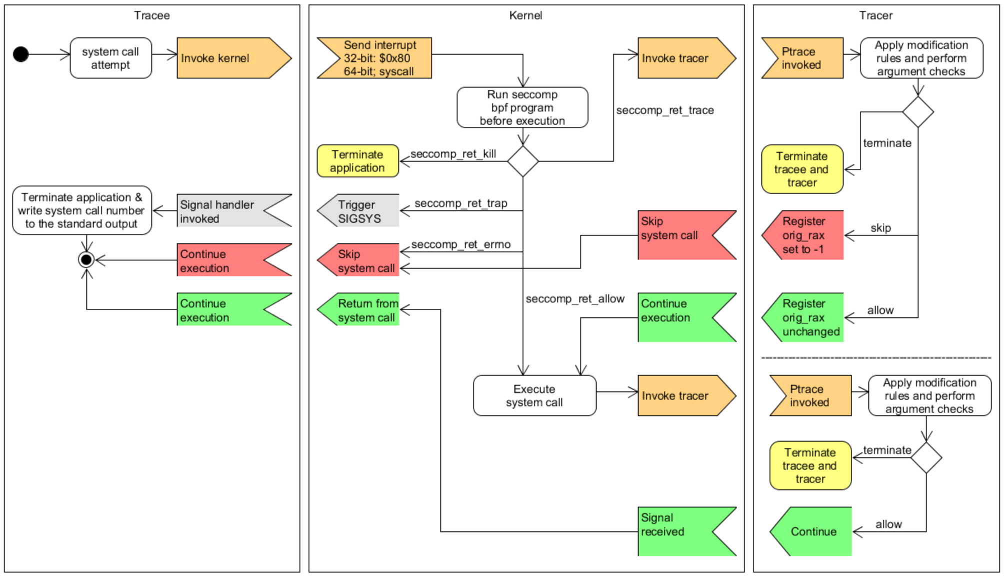

Fig. 2. System call execution in Downright

For example, the prototype for *getcwd*(2) looks like this:

```
/*
 * systemcall:          SYS_getcwd
 *
 * set_group[buf]:      path
 * set_length[buf]:     size
 */
 void sec_getcwd(__OUT char *buf,
                 unsigned long size) {
  char *cwd = getPidCwd(pid);
  strncpy(buf, cwd, size);
  free(cwd);

  OVERWRITE(buf, buf)
  OVERWRITE(return, strlen(buf))
  SKIP_SYSTEMCALL()

  CHECK_RULES()
}
```

Each configuration prototype begins with a comment block giving the system call's parameters various attributes. In this example, the `set_group` attribute tells Downright that the parameter `buf` in the system call definition is also referred to as `path` in the rules configuration file. It also informs Downright that `size` is the amount of space reserved for `buf`, something that one cannot say in ordinary C.

The new system call's header is located right below this comment block. The function's name must be different from that of the system call and the return value should be **`void`**. The arguments must be equal to the prototype of the system call. In this example, we use `__OUT` to signal that the parameter `buf` is a return parameter.

The wrapper function body then contains the actual implementation of the system call. Normally, this will make use of helper macros such as `OVERWRITE` (overwrites one parameter with data from another) or `CHECK_RULES` (inserts the generated rules at this point).

2) Security Rules: The framework provides an extensive language for defining access rules for system calls. The same syntax can also be used to define the rules that modify the parameters of a system call. The example configuration shown in Figure 3 demonstrates some of Downright's capabilities. The compilation process that takes these rules and compiles them into Seccomp-BPF and PTrace programs cannot be described within the confines of this paper; the reader is encouraged to read the source code on https://github.com/deforation/sec_seccomp_framework and to get in touch with the authors.

The **`[General]`** section determines the basic system call access rules for the tracer and its tracee. The `default_action` defines what action will be performed if a system call is made for which no rule exists:

- `terminate` causes the termination of the tracee invoking the system call.
- `skip` causes the kernel not to execute the system call. The error number in *errno*(2) will be set to `ENOSYS`.
- `allow` permits the execution of the system call.
- `trap` terminates the application and prints information about the responsible system call. This option makes more sense for debugging purposes on the tracer side.

A default action on the tracer side can be specified with the option `default_action_tracer:`. Using the option scheme `syscall <action>` as shown in lines 6 and 7, the access rights to system calls as a whole can be specified. Unlike the default actions, `trap` is invalid and `modify` can be used instead. This option forces Seccomp to always contact the tracer for checking and deciding on the final verdict. This option should be used with caution as it causes performance of system call execution to drop drastically (see Section V).

```
1  [General]
2  debug: False
3
4  # Client specific rules
5  default_action: terminate
6  syscall allow: exit, exit_group, close, fstat,
7    getrlimit
8  syscall modify: gettimeofday, getcwd
9
10 # Tracer specific rules
11 default_action_tracer: terminate
12 tracer allow: ptrace, wait4, getpid, socket, sendto,
13   read, chdir, getcwd, fstat, lseek, lstat, open,
14   close, kill, exit, exit_group, write, readlink,
15   connect, prlimit64
16
17 [Global]
18 path redirect: dir_contains("/invalid") => "/valid"
19
20 [open]
21 default: skip
22 path allow(r): dir_starts_with("./config")
23 allow(cw):
24   filename dir_starts_with("./create_write_only")
25 terminate(r): filename dir_starts_with("./logs"),
26   filename dir_starts_with("/bin")
27 redirect: path dir_ends_with(".dat") => ".txt"
28
29 [setrlimit]
30 default: allow
31 skip: resource == RLIMIT_CPU
32   && limit->rlim_max > getHardLimit(pid, resource)
33 redirect: limit->rlim_cur > limit->rlim_max:
34   limit->rlim_cur = limit->rlim_max
35
36 [chdir]
37 default: allow
38 path skip: not dir_starts_with("./")
39
40 [dup]
41 default: allow
42 fd skip: fd_path_starts_with("./secret")
43
44 [socket]
45 default: terminate
46 allow: domain == AF_UNIX && type == SOCK_STREAM,
47   domain == AF_LOCAL && type == SOCK_STREAM
48
49 [read:after]
50 default: allow
51 redirect: buf starts_with("not") => "its",
52   buf contains(" ") => " "
53
54 [write]
55 default: allow
56 buf redirect:  contains("i")=>"!",
57   contains("e")=>"3", contains("a")=>"4",
58   contains("l")=>"1", contains("t")=>"7",
59   contains("s")=>"5"
```

Fig. 3. System call rule demonstration example.

On the client side, the one system call needed by an application is *exit*(2) (or *exit_group*(2), depending on the standard C library used). On the tracer side, the particular system call actions can be defined, using the option scheme `tracer <action>`. For the tracer to be operable, more privileges such as *ptrace*(2), *wait4*(2) and others are required. The last option `debug:` specifies whether the framework is used in debug. If activated, all Seccomp actions are redirected to the tracer, which then logs information about the system call before the initial action is executed.

The **`[Global]`** section allows defining detailed rules that affect all system calls. The rule in line 18 shows that when accessing a path containing the string */invalid*, the path is modified and the occurence is replaced with */valid*. An application accessing the file */home/user/invalid/config.ini* will be redirected to */home/user/valid/config.ini*.

Other than the global sections, the rules for the specific system calls can be defined by creating a section named after the system call. The **`[open]`** section defines the rules for *open*(2). Each such section requires a `default:` attribute, defining the standard action for the system call. The first rule in line 22 determines that the files in the *config* directory of the current working directory can only be accessed in read-only mode. It is therefore not possible to create or write files in it. The subdirectory *create_write_only*, on the other hand, allows files to be created and written to. For the `terminate:` action, based on read permissions, two rules are defined. The first rule specifies that the application terminates when it tries to read data from the subdirectory *logs*. The second rule dictates that the application terminates if read attempts are made for the files in the */bin* directory. The last rule in line 26 shows a scenario where all attempts to open files ending in *.dat* are changed so that the corresponding *.txt* file is opened instead.

More complex rules can be established as shown in lines 31–34 for *setrlimit*(2). This example shows how to prevent an application, even one running as *root*, from increasing its CPU time limit: the application can only reduce the amount of time it can consume. The second rule in line 33–34 makes the system call fail for some resources, if the soft limit is greater than the hard limit. If this is the case, the soft limit is reduced to the same value as the hard limit.

The *chdir*(2) call changes the process's working directory. To prevent an application from changing its main directory to the root directory or any other sensitive directory, a rule as shown in line 38 can be deployed. This rule ensures that the tracee cannot move its working directory up within the directory hierarchy of its current working directory: it can only go deeper. This can drastically increase the security when combined with a similar rule for *open*(2), allowing the files to be opened within the current working directory only.

The rule in line 42 shows the limited capability of the possible checks concerning file descriptors. In this case, the descriptor path is evaluated and if it points to a file in *secret*, this rule will prevent the creation of a file descriptor copy.

The framework not only has the ability to work with system calls before their execution, but it also specifies the rules after they have been executed. Such a section must be marked with the attribute **`[syscall:after]`** after the name of the system call, as shown in line 51. The rule performs two actions. If the data read starts with “not”, it is replaced with “its”. The second rule replaces all the occurrences of double spaces with single spaces.

In the last example, we manipulate a buffer. Regardless of the location where the application attempts to write data, it gets modified. The six rules in lines 56–59 transform all output into leetspeak, also known as 1337. If an application runs the print command for the text “This is leetspeak”, what will appear instead is “Th!5 !5 13375p34k”. The example shows that all of the redirection rules to modify data are executed consecutively.

C. Build Process

In order to incorporate Downright into your own programs, follow these steps:

1) Download Downright from https://github.com/deforation/sec_seccomp_framwork
2) Place the folders *Generator* and *seccomp_framework* into the program’s main source directory.
3) Modify the file containing `main()`:
   - `#include "seccomp_framework/seclib.h"`
   - Rename the main function to `sec_main_after`.
   - Create a new main function and call
     ```
     return run_seccomp_framework(argc, argv,
       NULL, sec_main_after);
     ```
4) Add Downright files from *seccomp_framework* to the makefile so they are compiled together with the main application: *seclib.c*, *sec_client.c*, *sec_ptrace_lib.c*, *sec_seccomp_bpf_generator.c*, *sec_seccomp_rules.c*, *sec_syscall_emulator.c*, and *sec_tracer.c*.
5) Formulate security rules within the file *sec_rules.ini*. Additional modifications to the file *sec_syscalls_conf.c* may be required.
6) Run the configuration builder of Downright with the command
   ```
   python3 SecConfigBuilder.py \
     -o ../seccomp_framework
   ```
7) Compile and link the program.

V. Evaluation

A. Case Study: nginx

`Nginx` is a popular web server used by 63 % of the world’s busiest sites [24]. Security for these kinds of tools is an absolute necessity since they are exposed directly to the Internet. On the basis of different attack scenarios, and based on relevant CVEs [25], we build Downright configurations and evaluate their effectiveness.

1) Mitigating CVE-2016-1247: In 2016, a vulnerability was published that allowed local users with access to the web server user *www-data* to gain root access. All an attacker had to do was replace `nginx`’s error log with a symbolic link pointing to */etc/ld.so.preload*, which can be used to replace library functions and system call stubs with custom functions for the entire system.

```
[General]
syscall skip: symlink

[symlink]
default: allow
skip: path dir_starts_with("/etc")
```

Fig. 4. Downright fix for `nginx` vulnerability CVE-2016-1247.

A normal, unprivileged user can not create files in */etc*. But the process to rotate `nginx`’s logs runs as root and periodically replaces log files. If this happens, the symlinked error log gets created as */etc/ld.so.preload*, with write access for *www-data*. An attacker can then remove the symbolic link and write custom wrapper function to */etc/ld.so.preload*.

To mitigate this attack, it is necessary to know how an attacker can gain access to the user *www-data*. Since hosted pages and `nginx` run under this user, this means mitigating a remote code execution (RCE) vulnerability.

The example configuration in Figure 4 shows how this could be done. The first fix in line 2 revokes permission to *symlink*(2). Without access to this system call, an attacker is unable to create a symbolic link to */etc/ld.so.preload*. If the system call should be allowed, the fix from lines 4–6 can be used. The validation check tests if the target of the symbolic link starts with */etc*. If this is the case, the system call will not be executed.

Note that this vulnerability is extemely hard to mitigate in `nginx`’s source code, because a source code fix would have to mitigate *all possible* RCE vulnerabilities. However, if mitigating RCE vulnerabilities were easy, they wouldn’t exist. A framework like Downright is therefore a very good second line of defense. Note also that preventing *symlink*(2) could have been done not as a mitigation, but as the result of an analysis before release, just as is normally done to create an AppArmor profile. In this case, the mitigation would have been elevated to a prevention.

2) Mitigating CVE-2009-2629: This was one of the most serious vulnerabilities `nginx` has had in its history. Attackers could execute arbitrary code through specially crafted HTTP-requests. In certain cases, this led to a buffer underflow and data of the uniform resource identifier (URI) was written to the heap before the allocated buffer. Such bugs cannot be detected on the system call level and their complete prevention using this framework is not possible.

However, Figure 5 shows a mitigation strategy. The security rules are configured so that `nginx` can only call a minimal set of system calls. Monitoring `nginx` reveals that the following system calls, and only those, are needed to host HTML pages: *accept4*, *bind*, *brk*, *clone*, *close*, *connect*, *dup2*, *epoll_create*, *epoll_ctl*, *epoll_wait*, *eventfd2*, *exit*, *exit_group*, *fcntl*, *fstat*, *geteuid*, *getdents*, *getpid*, *getrlimit*, *ioctl*, *listen*, *lseek*, *lstat*, *mkdir*, *mmap*, *prctl*, *pread64*, *pwrite64*, *read*, *recvfrom*, *rt_sigaction*, *rt_sigprocmask*, *rt_sigsuspend*, *sendfile*, *set_robust_list*, *setrlimit*, *setsockopt*, *shutdown*, *socket*, *socketpair*, *stat*, *uname*, *write*, and *writev*.

```
[General]
default_action: terminate
syscall allow: accept4, bind, brk, clone, close,
  connect, dup2, epoll_create, epoll_ctl, epoll_wait,
  eventfd2, exit, exit_group, fcntl, fstat, geteuid,
  getdents, getpid, getrlimit, ioctl, listen, lseek,
  lstat, mkdir, mmap, prctl, pread64, pwrite64, read,
  recvfrom, rt_sigaction, rt_sigprocmask,
  rt_sigsuspend, sendfile, set_robust_list,
  setrlimit, setsockopt, shutdown, socket,
  socketpair, stat, uname, write, writev

[setrlimit]
default: terminate
allow: resource == RLIMIT_NOFILE

[mkdir]
default: terminate
allow: path dir_starts_with("/var/nginx/tmp")
```

Fig. 5. Downright fix for `nginx` vulnerability CVE-2016-1247.

To further reduce the attack surface, access to these system calls should only be granted if the arguments fulfill certain criteria. For example, *setrlimit*(2), should only be allowed if the first argument equals `RLIMIT_NOFILE` because no other resource limits are ever changed by `nginx`. Similarly, *mkdir*(2) should also be restricted: `nginx` uses *mkdir*(2) to create five folders for temporary files in one specific directory.

Thus mitigated, an attacker confronted with such a limited set of system calls will have a harder time trying to seriously damage the system or steal confidential information, especially if validation checks to the most dangerous system calls *brk*(2), *mkdir*(2), *open*(2), *socket*(2), *ioctl*(2), *brk*(2), *dup2*(2), *setrlimit*(2), and *clone*(2) are applied [26].

However, mitigation for this vulnerability is better be done through AppArmor or SELinux, since no request broker architecture is needed for this kind of blanket allow/deny policy.

3) Mitigating CVE-2009-3898: This vulnerability, discovered in In 2009, allowed remote authenticated users to create and overwrite arbitrary files. The attacker only needed to change the header of a HTTP-request and change its destination to a path that contains *../*. To mitigate the vulnerability, multiple methods exist: access to unused files can be prevented by performing sanity checks on *open*(2) or, alternatively, it can be checked whether the request string contains *../*. The latter method became the publicly available fix.

The configuration file extract in Figure 6 contains both variants of the fix for the vulnerability. The first fix in lines 1–15 minimise the access permissions for files. The *open*(2) system call will be skipped unless `Nginx` reads its configuration files or files that belong to the web page. Write permissions are only given to log files and the application status file called *nginx.pid*. In addition, `nginx` is required to have permissions to create and read the status file. Note that the security rules use relative paths. Calling *chdir*(2) to change the working directory, should, be disallowed or absolute paths should be considered instead. The second fix, shown in lines 10–13, scans each request to the web server for the occurrence of the string *../*, which is then removed.

```
[General]
syscall skip: chdir, openat

[open]
default: skip
filename allow(r): dir_starts_with("./html"),
    dir_starts_with("./conf")
filename allow(w): dir_starts_with("./logs"),
    dir_starts_with("./access.log")
filename allow(rwc):
    dir_starts_with("./logs/nginx.pid")

[recvfrom:after]
default: allow
buf redirect: contains("../") => ""
#redirect: buf contains("../"): return => -1
```

Fig. 6. Downright fix for `nginx` vulnerability CVE-2009-3898.

As much as one wuld like to terminate the process if *../* is encountered in the request, one has to concede that the request itself is valid, since there is nothing in any standard that would forbid it. Another solution would be to replace the return value of the system call with `-1`. `Nginx` then thinks that the system call failed and the vulnerable source is invoked. This is the commented-out solution in line 16.

4) Conclusion: When no request broker is needed, mitigations should not be done within Downright. While mitigating `nginx` issues with Downright is not trivial, it is surprising how small the configuration files are relative to their impact. Note also that mitigating some security issues cannot be done inside `nginx`'s source code. For example, to ensure that *mkdir*(2) is not called outside a given directory, it is not enough to check every *mkdir*(2) call in the `nginx` source code, since the entire issue is mitigating these problems if, e.g., *mkdir*(2) is called through an RCE exploit and not from `nginx` code.

B. Performance Evaluation

1) Performance Overhead Per System Call: We used a Debian platform consisting of a virtual machine running on an Intel Core i7-4700HQ at 2 GHz with two cores assigned. Each system call was executed five million times in a row on five different configurations:

- Without framework integration.
- Permit execution using a Seccomp rule.
- Disallow execution using a Seccomp rule.
- Invoke the tracer using a Seccomp rule and permit the execution of the system call.
- Invoke the tracer using a Seccomp rule and disallow the execution of the system call.

Figure 7 shows the result of the performance measurements.

The *getrlimit*(2) system call is often used to obtain information about certain resource restrictions such as the number of allowed processes, the available execution time, maximal stack size and others. The overhead for the Seccomp-based checks is between 46–63 %. Decisions using the tracer usually take about 108 times longer than an immediate execution.

The *open*(2) and *close*(2) system calls were also tested in a much more realistic scenario, in which a file was repeatedly

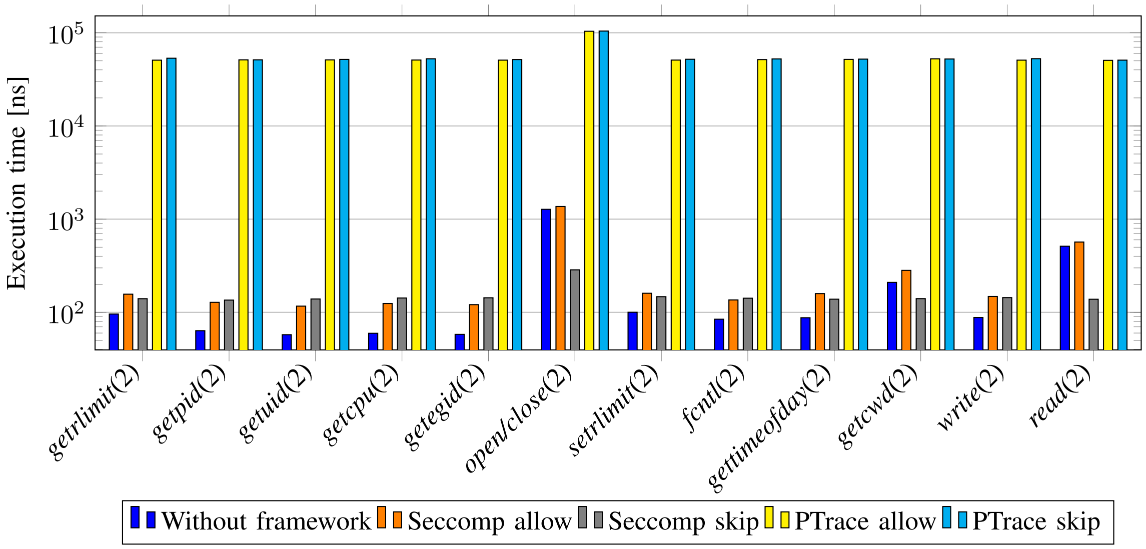

Fig. 7. Result of performance measurements.

opened and closed. For these system calls Seccomp causes an overhead of only 7.2 %, which is a good result because they are usually called multiple times within an application. If the tracer is also used, the performance overhead rises to 16.2%.

This is in apparent conflict to Figure 7, which shows a much higher performance penalty. The solution is that, even though executing the system call with the tracer results in a hundredfold increase in execution time, in reality, the real time between the call to *open*(2) and its return is not spent mostly executing kernel code. Rather, it is spent waiting for I/O to complete. This is in contrast to, e.g., *getrlimit*(2), where information from the process descriptor is copied into the result.

All this is good news because it suggests that in ordinary workloads, the massive increase in system call execution time due to the tracer will largely go away. This is borne out in our performance study on `nginx`; see the next section.

2) Performance Overhead For `Nginx`: In our scenario, `nginx` is used to host a small site consisting of static content and the functionality to traverse directories using the integrated autoindex module. The number of requests that `nginx` can process with and without Downright is measured by `httperf` [27], which allows opening multiple connections to a web server, logging request service times.

Sending 700,000 requests to seven resources in 100,000 sessions took `nginx` 50.66 s to service (13,816.6 requests per second). The same scenario on `nginx` protected by the Downright rules defined in Figure 8 took 109.32 s (6,403.2 requests per seconds). Performance is therefore about halved.

It should be noted, however, that every GET request needs to go to the tracer, which is very expensive. In this case, we did this to showcase Downright's abilities for complex system call parameter and result manipulation, but if the only reason the tracer is invoked is to forbid ../ in requests, this can more easily be handled inside `nginx` itself: if ../ is used to access files outside the virtual server's web root, then the protections for *open*(2) will effectively prevent that also.

C. Performance Optimisation

As we have seen in the previous sections, Downright's performance overhead is not negligible. This is obviously one of the major stumbling blocks for people to use Downright.

As we saw in Figure 7, whenever PTrace is involved, performance suffers. This is first because invoking PTrace means at least two additional context switches, which are expensive. Luckily, PTrace is only needed for the fancier kinds of system call manipulation, such as manipulating system call arguments or return values. This should be rare, so this heavy performance hit should not normally occur. (In the case of `nginx`, one reason for the large performance hit was due to manipulation of arguments to system calls like *mkdir*(2), which could also be accomplished using *chroot*(2) jails.) The second reason why PTrace is expensive is because the broker itself may need to make system calls of its own. One possibility to reduce this high number of user-to-kernel crossings is to incorporate Downright's tracer into the kernel. This should eliminate the largest obstacle to higher speeds.

Another reason for Downright's performance overhead is the behaviour of PTrace on multithreaded applications. As explained in Section III, PTrace will stop all threads in a multithreaded application so that examination of the executing application can finish without having to consider the possiblity of data races. While this guarantees the correct execution of the tracee, it obviously hinders performance.

The key insight here is that, for the purposes of Downright, threads may continue while a system call is processed by Downright. If an application has a data race while a system call is in progress, that application is buggy. Allowing data races to occur during system calls therefore does not make the

```
[General]
debug: False

default_action: terminate
syscall skip: chdir, chroot
syscall allow: exit, exit_group, close, fstat, lstat,
  lseek, getpid, pread64, epoll_create, brk, geteuid,
  mkdir, ioctl, getrlimit, setrlimit, stat, fcntl,
  setsockopt, listen, mmap, rt_sigprocmask, accept4,
  rt_sigaction, clone, pwrite64, dup2, socketpair,
  rt_sigsuspend, set_robust_list, prctl, eventfd2,
  epoll_ctl, epoll_wait, bind, sendfile, shutdown,
  writev, write, read, uname, socket, connect
default_action_tracer: terminate
tracer allow: ptrace, wait4, getpid, sendto, chdir,
  getcwd, lseek, lstat, readlink, kill, connect, fstat

[open]
default: skip
filename allow(r): dir_starts_with("./html"),
  dir_starts_with("./proc/stat"),
  dir_starts_with("/conf"),
  dir_starts_with("/proc/cpuinfo"),
  dir_starts_with("/etc/localtime"),
  dir_starts_with("/sys/devices/system/cpu/online"),
filename allow(w): dir_starts_with("./logs"),
  dir_starts_with("./access.log")
filename allow(rwc): \
  dir_starts_with("./logs/nginx.pid")
filename redirect: dir_ends_with(".txt") => ".dat"

[getdents]
default: allow
fd skip: fd_path_contains("nolist")

[recvfrom:after]
default: allow
redirect: buf contains("../"): return => -1,
  buf starts_with("GET /data/private/") \
     => "GET /data/fake_private/"
```

Fig. 8. Nginx case study: rule set for performance measurements.

application any buggier than it already was.[7] At the moment, Downright itself is not multithreaded. Enabling the process to continue other threads while a single thread is stopped by PTrace, and enabling Downright to process several system calls at once would improve Downright's performance again.

When Seccomp alone is being used, performance overheads are roughly between 7–50 %. The reason for this large range is that purely informative syscalls that simply copy a value from the process descriptor, like *getpid*(2), are impacted most severely. System calls like *open*(2), *read*(2), or *write*(2) may get by with comparatively little impact of about 7 %, as we saw above. This is actually good news, since the calls that are actually responsible for most of the heavy lifting in server processes are those I/O calls, and they have comparably little overhead. The solution here would be to simplify the checking that the rules do, and to optimise these cases specifically.

As we saw in the `nginx` case study, an I/O-heavy process that uses PTrace may be only half as fast as an unprotected one. But note that protecting nginx with these rues drastically reduces the attack surface available to an attacker, so one might argue that this may well be a price worth paying.

To summarise, these are the most promising areas for performance improvement:

- use Seccomp as much as possible;
- incorporate Downright's tracer into the kernel to reduce the large number of context switches; and
- continue executing threads while a single thread is stopped by PTrace.

One good usability performance optimisation would be to include the initialisation of Downright's runtime in a specialised *crt0.o*. This is the file that contains the linked program's main entry point. It is responsible for initialising the C runtime and for calling `main()`. If such a *crt0.o* would be written, then Downright-enabled programs could be built without changing *any* source code whatsoever.

VI. Conclusion

We have introduced Downright, a framework and toolchain for privilege handling in Linux and comparable Unix-like systems. Downright's toolchain consists of a system call prototype file, to be written once per system call, and an application-specific rule set, to be written separately for each application. Once written, inclusion of Downright into the application is easy, requiring a one-line modification to `main()`.

Once the process runs, all its system calls are automatically routed through a combination of Seccomp and PTrace (if configured). The rules defined in the rule set allow for very fine-grained control over what system calls are allowed when, and even allow manipulating parameters and return values.

Performance was tested for `nginx`, where an I/O heavy test load that triggered the most expensive rule evaluation revealed that performance drops to roughly half of the performance of an unprotected `nginx`.

Downright is suited for programs that need special protection and that must run on Unix-like systems on which protection from other mechanisms such as AppArmor or SELinux may not be available, e.g., when the person running the program does not have root access to install AppArmor or SELinux profiles, or when reboots to activate SELinux profiles cannot be countenanced.

Of course, there are security situations for which Downright is not especially suited. For example, if one wants to allow the creation of child processes generally, but limit the *number* of child processes, Downright is the wrong tool and `setrlimit(RLIMIT_NPROC, ...)` should be used instead. Downright should really only be used to limit a process's access to system calls and, in exceptional circumstances, to modify a system call's arguments and return values.

We do not wish to give the impression that Downright is a cure-all. It is not. But we are excited by the possibilities of new techniques like Seccomp-BPF and urge the community to consider experimenting with them in order to provide what is at the moment the largest obstacle for their adoption: better tooling. We believe that Downright makes an important contribution to that end.

[7] It might, however, make it easier to trigger the bugs with malicious requests, so this argument is advanced only cautiously.

VII. Acknowledgments

The authors would like to thank Felix von Leitner, whose talk at the 23c3 [28] and blog post on Seccomp in Firefox [29] gave rise to this work.

We would also like to thank the reviewers, especially Reviewer Three, whose numerous constructive comments made this a much better paper.

References

[1] "Linux sandboxing," Jan. 2019. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md

[2] Android Developer's Blog, "Seccomp filter in android o," Jan. 2019. [Online]. Available: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

[3] Docker Documentation, "Seccomp security profiles for docker," Jan. 2019. [Online]. Available: https://docs.docker.com/engine/security/seccomp/

[4] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th Usenix Security Symposium*. Berkeley, CA, USA: USENIX Association, 2003, pp. 257–272. [Online]. Available: https://www.usenix.org/legacy/events/sec03/tech/provos.html

[5] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious server security," in *Proceedings of the 14th Systems Administration Conference (LISA 2000)*. Berkeley, CA, USA: USENIX Association, Dec. 2000, pp. 355–367. [Online]. Available: https://www.usenix.org/conference/lisa-2000/subdomain-parsimonious-server-security

[6] C. Brown, *SUSE Linux*. Sebastopol, CA, USA: O'Reilly Media, Jul. 2006, ISBN 059610183X.

[7] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," National Security Agency, Tech. Rep., Feb. 2001. [Online]. Available: https://www.nsa.gov/resources/everyone/digital-media-center/publications/research-papers/assets/files/flexible-support-for-security-policies-into-linux-feb2001-report.pdf

[8] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2003, pp. 231–242. [Online]. Available: https://www.usenix.org/legacy/events/sec03/tech/provos_et_al.html

[9] N. Provos. (2002, Mar.) Privilege separated openssh. [Online]. Available: http://citi.umich.edu/u/provos/ssh/privsep.html

[10] M. Hafiz, R. E. Johnson, and R. Afandi, "The security architecture of qmail," in *Proceedings of the 11th Conference on Pattern Languages of Programs*, Sep. 2004.

[11] D. J. Bernstein. (1997, Mar.) The qmail security guarantee. [Online]. Available: https://cr.yp.to/qmail/guarantee.html

[12] D. G. Murray and S. Hand, "Privilege separation made easy: Trusting small libraries not big processes," in *Proceedings of the 1st European Workshop on System Security*. New York, NY, USA: ACM, 2008, pp. 40–46. [Online]. Available: http://doi.acm.org/10.1145/1355284.1355292

[13] J. Wang, X. Xiong, and P. Liu, "Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications," in *Proceedings of the 2015 USENIX Annual Technical Conference*. Santa Clara, CA: USENIX Association, 2015, pp. 361–373. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/wang_jun

[14] D. Kilpatrick, "Privman: A library for partitioning applications," in *Proceedings of the 2003 USENIX Annual Technical Conference, FREENIX Track*, Jun. 2003, pp. 273–284. [Online]. Available: https://www.usenix.org/legacy/events/usenix03/tech/freenix03/kilpatrick.html

[15] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 57–72. [Online]. Available: https://www.usenix.org/legacy/events/sec04/tech/brumley.html

[16] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 323–333. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693091

[17] M. Trapp, M. Rossberg, and G. Schaefer, "Automatic source code decomposition for privilege separation," in *Proceedings of the 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2016, pp. 1–6.

[18] M. Kerrisk, *The Linux Programming Interface: A Linux and Unix Programming Handbook*. No Starch Press, 2010.

[19] H. Chen, D. Wagner, and D. Dean, "Setuid demystified," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 171–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=647253.720278

[20] E. Allman and B. Costales, *sendmail*. Sebastopol, CA, USA: O'Reilly Media, Nov. 1993.

[21] I. R. Jenkins, S. Bratus, S. Smith, and M. Koo, "Reinventing the privilege drop: How principled preservation of programmer intent would prevent security bugs," in *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*. New York, NY, USA: ACM, 2018, pp. 3:1–3:9. [Online]. Available: http://doi.acm.org/10.1145/3190619.3190635

[22] IOVisor Project. (2019, Jul.) BPF compiler collection (BCC). [Online]. Available: https://github.com/iovisor/bcc

[23] J. Edge. (2015, Sep.) A seccomp overview. [Online]. Available: https://lwn.net/Articles/656307/

[24] Nginx, Inc. (2018) nginx. [Online]. Available: https://www.nginx.com/?_ga=2.228374173.103440784.1525847183-1521241523.1523906676

[25] MITRE Corporation. (2018) CVE details—the ultimate security vulnerability datasource. [Online]. Available: https://www.cvedetails.com/

[26] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating system enhancements to prevent the misuse of system calls," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2000, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/352600.352624

[27] D. Mosberger, M. Arlitt, T. Bullock, T. Jin, S. Eranian, R. Carter, A. Hately, and A. Chadd. (2018) httperf. [Online]. Available: https://github.com/httperf/httperf

[28] F. von Leitner. (2015, Dec.) Check your privileges. Talk at 32c3. [Online]. Available: https://www.youtube.com/watch?v=2e91cEzq3Us

[29] ——. (2017, Aug.) Ich baue mir unter Linux meinen Firefox ganz gerne selbst. [Online]. Available: https://blog.fefe.de/index.html?ts=a77b2171