Sandboxing With Seccomp And Seccomp-BPF

Michael Eder

Technische Universität München
Fakultät für Informatik
`edermi@in.tum.de`

1 July 2018

Abstract. Seccomp is a system call filtering mechanism available in modern Linux kernels. In conjunction with the extended Berkeley Packet Filter (eBPF) available in the kernel, it is a powerful mechanism to filter for syscalls and perform actions like logging or killing processes on specific events. This work aims to give an overview on what seccomp is and how it evolved over the time. There are some brief examples on how to use seccomp for system call filtering. Some applications and higher level abstractions as well as alternative frameworks will also be presented.

Keywords: seccomp · BPF · secure computing · container · sandbox

1 Introduction

On modern operating systems, system resources are usually managed by the operating system kernel. Processes that like to access and interact with resources are not able to do this directly because this may negatively impact other processes or the system at all and every program had to bring its own functionality to perform those operations (e.g. network driver functionality to be able to do networking). Instead, the operating system provides abstraction layers and interfaces to access such functions, called *system calls* or short *syscalls*. A process that wants to perform an action, for example reading and writing to a network resource or a file, can request access to the resource via the *open(2)* syscall. The operating system is able to do all safety and security checks and after performing required steps to make the resource available, the process gets a file descriptor that can be used with *read(2)* and *write(2)* syscalls. After doing the work, *close(2)* closes the file descriptor and the operating system does some cleanup like syncing with storage or cleaning up memory. Usually, there are several hundred syscalls and some of them can be abused, e.g. an attacker may use *execve(2)* to spawn a shell.

In 2005, Andrea Arcangeli introduced a feature called *secure computing*, short *seccomp*, to the Linux kernel [3]. The goal was to enable simple resource sharing by allowing to run untrusted code on a system and restricting it to a predefined set of system calls that may be issued - apart from performing computations, just enough to communicate over previously provided file descriptors, returning from computation or exiting the program.

In 2009, discussions arose if the feature is actually used or maybe should be abandoned [3]. Developers of the Chromium project stated that they use seccomp for building a browser sandbox, but they were suffering from the limited functionality and instead of removing the feature, people started discussing on how to improve seccomp and where to direct the feature. In 2012, the new seccomp mode 2 was added to the kernel. It allowed developers to use a virtual machine called the Berkeley Packet Filter (BPF) which already resided in the kernel. It is primarily used for network packet filtering, but now it was also able to apply filter rules on system calls and their arguments.

This combination allows to implement flexible and fine grained sandboxes as well as to harden existing applications a bit at a time. Today, seccomp is used in several applications and there are some interesting higher-level abstractions that further ease the usage.

The remainder of the paper is structured as follows: Section 2 gives an overview over the original secure computing implementation whereas Section 3 explains BPF, the new filter capabilities, how to use seccomp-BPF by hand and by using the libseccomp library. Section 4 gives an overview over other security related Linux kernel features and explains the differences and relationships to seccomp. Some information that helps to assess the security of seccomp is given in Section 5 while Section 6 gives an overview of some projects using seccomp. Section 7 presents related work in form of similar approaches for Linux and other operating systems. Last but not least, Section 8 gives a conclusion.

2 Reducing kernel attack surface with seccomp

The original seccomp implementation aimed to prohibit the usage of all system calls except *read* (2), *write (2)*, *exit (2)* and *sigreturn (2)* [3]. This way, a process can be locked down in a way that it is still possible to do computations and communicate using provided file descriptors, but all other actions apart from exit or return will lead to a `SIGKILL` signal by the kernel, meaning that the process is immediately killed. A developer that wants to enhance the security of his code, e.g. a web server, can open all required resources during an initialization step and afterwards use seccomp to drop the privilege to call system calls apart from the ones above. The web server will still be able to do computations and serve requests while any attempt to interact with resources that were not made available during initialization will fail.

This mode is still available and the operation value signaling that it should be used is `SECCOMP_SET_MODE_STRICT` [5]. Listing 1.1 shows a basic program that uses the strict seccomp mode. It also illustrates how to use file descriptors in order to communicate. In line 9 the program checks if there was an additional parameter. If yes, the parameter will be printed on `stdout`. Afterwards, seccomp is enabled and the allowed system calls are therefore restricted. The most interesting things here are probably in line 13 and 14. The `printf` will write a string to the standard output. If the standard output was already opened, e.g. by an earlier `printf`, this will succeed and print the *“Can you see me?”* message. If the standard output is still closed because the first `printf` was not executed, the attempt to open it will call an illegal system call (*fstat(2)* in order to get the status of the file descriptor) and the kernel will kill the program. Interestingly, this also happens by ending the program with a `return` statement, hence the direct system call to *_exit(2)*. The seccomp manpage [5] states clearly that "*_exit(2)* (but not *exit_group(2)*)" are allowed. Since version 2.3 of glibc, *exit_group(2)* is used to end processes [6], therefore the kernel will kill the process even when trying to make a clean exit. This makes sense, because *exit_group (2)* ends all threads in a process, but a thread should not be able to influence the behavior of other threads, not even killing them. A more elaborate explanation on this behavior can be found in [7].

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <sys/prctl.h>
 5 #include <linux/seccomp.h>
 6 #include <sys/syscall.h>
 7
 8 int main(int argc, char *argv[]) {
 9         if (argc == 2) {
10                 printf("%s\n", argv[1]);
11         }
12         prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
13         printf("Can␣you␣see␣me?\n");
14         syscall(SYS_exit, EXIT_SUCCESS);
15 }
```

Listing 1.1. A simple example for the seccomp strict mode

3 System call filtering using seccomp and BPF

The idea of seccomp is interesting and makes it possible to introduce strong boundaries when it comes to abusing hijacked processes but it was barely used because it enforces a specific computing model with no room for flexibility, e.g. giving a process the opportunity to interact with the environment apart from file descriptors opened beforehand. This inflexibility also made it hard for developers to port their software to work with seccomp. Lots of libraries are not compatible with the hard enforcement of predefined syscalls, c.f. Listing 1.1 Line 14 where it was required to do a syscall by hand instead of using the *exit(3)* wrapper provided by glibc. Furthermore it is not possible to do a steady transition by locking parts of programs down syscall for syscall.

In 2009, the question arose if the seccomp feature was actually used at all [3]. As it turned out, Google started to use it for their Chromium browser and discussions started how to improve the feature in a way that it is more flexible but still maintainable and reasonably fast. In 2012, the patches for "seccomp mode 2" were submitted [4] and seccomp was able to perform syscall filtering according to a policy expressed by a *Berkeley Packet Filter (BPF)* program.

3.1 Berkeley Packet Filter (BPF) and extended Berkeley Packet Filter (eBPF)

The Berkeley Packet Filter was originally built in the early 1990ies to filter for network packets [16]. The goal was to build a general, protocol independent virtual machine based packet filter which had instructions that were easy to decode, had minimal packet references in order to minimize memory accesses and the virtual machine's registers should reside in physical registers. This virtual machine resides in the operating system kernel and significantly improved packet filtering performance compared to other approaches at that time. The code executed by the machine is some filter expression which may also be user-supplied. The original concept was extended by a Just in Time (JIT) compiler and in 2014 a redesign was committed to the kernel [17] which increased performance by adapting the virtual machine to modern architectures and its features, e.g. more and broader registers.

This *extended Berkeley Packet Filter (eBPF)* was at the beginning only used internally in the kernel but soon got accessible from userspace [1]. Loading user-supplied bytecode into a virtual machine living inside the operating system kernel sounds like a questionable idea, so there is a verifier that checks for some properties of the loaded code before and while running it [1]:

- Loop-free: The verifier ensures that there are no loops in the program which is important in order to make sure that the program will terminate. This is done by constructing a control flow graph and performing a depth-first search on it. This leads also to another important property: the program can not be Turing complete which is an important property from the language security point of view.
- There are no unreachable instructions.
- Valid stack and register states after each instruction. This prevents access to data that is out of range or jumps to addresses that are out of bounds.
- No pointer arithmetic in "secure mode". This means that all code loaded by a non-superuser must not contain pointer arithmetic in order to prevent memory leakage to unprivileged users. Privileged users are allowed to perform pointer arithmetic, but all operations are still checked for type, alignment and bounds violations.
- Uninitialized stack memory or never-written-to registers can not be read and writing to read-only stack memory or registers is prevented.
- The BPF program type determines which kernel API and memory is exposed to the BPF program, "For example, a tracing program does not have the exact same subset of helper functions as a socket filter program (though they may have some helpers in common). Similarly, the input (context) for a tracing program is a set of register values, while for a socket filter it is a network packet." [18]

The LLVM project provided a compiler in order to emit eBPF assembly, so users weren't forced to write the bytecode on their own anymore. Furthermore, projects like *bcc (BPF compiler collection)* make it possible to compile eBPF bytecode outside of the kernel source tree, which is one of the downsides when using LLVM.

For the remainder of the paper, *BPF* in the context of seccomp refers to *eBPF* as well.

3.2 BPF and seccomp

Since BPF was built with extensibility in mind, it is also possible to use it for different purposes than filter network data. For example it is possible to write programs that observe and monitor other processes or even the kernel itself. This makes debugging or performance optimization easier since an BPF program can be loaded dynamically on the fly. The repository of *bcc* [19], a standalone compiler for BPF, contains over 100 examples and tools using various programming languages that demonstrate what can be done with BPF.

BPF can also be used in conjunction with seccomp. This makes seccomp a lot more flexible and opens new possibilities. Instead of killing a process if it calls any other syscall than one of the four allowed ones, it is possible to apply a black- or white-listing approach. For example, a developer may want to prevent any call to *execve(2)* and allow any other syscall. This may be desired for example to provide a quick hot fix for an already publicly exploited vulnerability until a more elaborate fix is available. Also, a blacklist approach is good at getting started with sandboxing, especially for larger code bases like browsers. Shutting syscalls down one after another and seeing what breaks enables a steady transition to a sandboxed program. Generally the white-listing approach is considered more secure because it is more likely to accidentally miss something evil in a large black list. The white list simply denies everything that was not explicitly considered legal for a specific reason.

Apart from black- or white-listing system calls BPF allows further to filter on system call arguments. This way, a process may for example be allowed to write to `stdout` but not any other file descriptor, giving developers fine grained control for specifying desired behavior. To make this possible, every filter gets access to a `seccomp_data` struct as shown in Listing 1.2 [5].

```
struct seccomp_data {
    int   nr;                    /* System call number */
    __u32 arch;                  /* AUDIT_ARCH_* value
                                    (see <linux/audit.h>)*/
    __u64 instruction_pointer;   /* CPU instruction
                                    pointer */
    __u64 args[6];               /* Up to 6 system call
                                    arguments */
};
```

Listing 1.2. struct seccomp_data

The arch field is necessary in order to be able to run userspace code for different architectures (e.g. i386 and x86_64) and is required to identify the system call correctly because they may have different numbers on different architectures. The instruction pointer value points to the memory of the process that issued a system call which may be interesting for debugging purposes. The args array contains up to six parameters passed to the syscall for inspection. Apart from the simple structure, there are some tricky caveats, for example the arch field is not sufficient to distinguish all architectures and the args field may contain data that is not passed to a syscall at all - the manpage [5] goes into detail about these issues.

Problems arise from the fact that every system call in Linux has a system call number which may differ on different architectures. For example, the *seccomp(2)* system call has the ID 317 on x86_64 [9] and the ID 354 on i386 [8]. The decisions of seccomp are based on syscall numbers. In the case that for example the libc wrappers behave differently, e.g. because of another libc implementation, a different version or a different architecture, seccomp filters may also behave differently which requires additional testing. This problem can lead to situations where the filters may be completely bypassed. For example x86_64 is able to run x86 code, too. A filter blacklisting certain syscalls may be bypassed by calling the respective 32bit system calls.

Seccomp returns a value after syscall inspection describing what is done (*SECCOMP_RET_ACTION_FULL*) [5]:

– *SECCOMP_RET_KILL_PROCESS* Kills the process immediately without executing the syscall. A coredump is written. This is default behavior since Linux 4.14 for all values other than the specified ones.
– *SECCOMP_RET_KILL_THREAD* Kills the thread immediately without executing the syscall, other threads continue execution. A coredump is written since 4.11 if the process is single-threaded. This was the default behavior before Linux 4.14 for all values other than the specified ones.
– *SECCOMP_RET_TRAP* The syscall is not executed. A signal is sent to the thread containing meta data like syscall number, architecture etc. The thread may resume operation.
– *SECCOMP_RET_ERRNO* The syscall is not executed and the return value of the seccomp filter is passed to the user space as *errno*.
– *SECCOMP_RET_TRACE* Notify a ptracer and give control to it. This is interesting because in Linux before 4.8 the result of the ptracer was not checked again which allowed to bypass the sandbox under some circumstances.
– *SECCOMP_RET_LOG* The syscall is executed and an event log entry is generated.
– *SECCOMP_RET_ALLOW* The syscall is executed.

The ability of seccomp in conjunction with BPF to define specific actions and perform logging makes it easier to debug and port applications to use these filters. Existing applications can be run while logging all syscalls in order to get an overview which syscalls are used in which places. Afterwards, they can be migrated step by step. The ability to pass control over to a dedicated ptracer allows to understand and debug more subtle bugs.

From now on, the term *seccomp* refers to both, traditional seccomp and using it in conjunction with BPF.

3.3 Practical use

The seccomp manpage [5] contains an example that takes a syscall, an architecture, a program as parameter and makes sure that the program only runs on the chosen architecture and that the syscall provided is not allowed. The whole program breaks the mold of this work, so we focus on the more interesting part, the filter, as depicted in Listing 1.3. The filter is an array of BPF instructions which are wrapped by functions defined in **linux/filter.h**. The filtering is done completely by hand using the BPF virtual machine almost directly. This requires that the programmer brings knowledge of the BPF virtual machine and writes the assembly completely on his own. Building more complex programs gets time consuming, hard to maintain and error prone.

```
struct sock_filter filter[] = {
  /* [0] Load architecture from 'seccomp_data'
      buffer into accumulator */
  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
      (offsetof(struct seccomp_data, arch))
  /* [1] Jump forward 5 instructions if architecture
      does not match 't_arch' */
  BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, t_arch, 0, 5
  /* [2] Load system call number from 'seccomp_data'
      buffer into accumulator */
  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
      (offsetof(struct seccomp_data, nr))),
  /* [3] Check ABI - only needed for x86-64 in blacklist
     use cases. Use BPF_JGT instead of checking against
     the bit mask to avoid having to reload the syscall
    number. */
  BPF_JUMP(BPF_JMP | BPF_JGT | BPF_K, upper_nr_limit, 3, 0
  /* [4] Jump forward 1 instruction if system call number
     does not match 'syscall_nr' */
  BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, syscall_nr, 0, 1
  /* [5] Matching architecture and system call: don't
     execute the system call, and return 'f_errno'
     in 'errno' */
  BPF_STMT(BPF_RET | BPF_K,
      SECCOMP_RET_ERRNO | (f_errno & SECCOMP_RET_DATA)
  /* [6] Destination of system call number mismatch: allow
      other system calls */
  BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW
  /* [7] Destination of architecture mismatch: kill task */
  BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
};
```

Listing 1.3. Excerpt of the filter rules of the seccomp example program from seccomp(2)[5]

The libseccomp library [23] provides higher level abstractions that can be used to generate filters which then can be passed to seccomp. It is platform independent and provides Go bindings as well. Listing 1.4 shows an example usage of libseccomp with C. Line 11 of the example defines a default policy that is applied for every syscall that has no other filter in place. In this case, the process should be killed. The example simply prints "Hello World!" via *write(2)* on stdout. This is possible because lines 19–21 add a seccomp filter rule that explicitly allows to use the syscall if its first argument, the file descriptor, equals the file descriptor of stdout (usually 1). Writing to any other file descriptor, e.g. a socket or stderr, will fail, hence this exemplary part is commented out in line 45. In contrast to the earlier example of the strict mode filtering, it is now also possible to do a clean exit on returning from the main function by allowing *exit_group(2)*.

Apart from the better readability and maintainability, using libseccomp has some further advantages, for example the manpage of *seccomp_rule_add(3)* [24] states that "the *seccomp_rule_add()* and *seccomp_rule_add_array()* functions will make a "best effort" to add the rule as specified, but may alter the rule slightly due to architecture specifics (e.g. internal rewriting of multiplexed syscalls, like socket and ipc functions on x86)". The library also takes care of resolving syscall numbers through a wrapper as can be seen in line 20 of Listing 1.4. This makes it easier for developers to write code that is portable across different architectures and less error-prone due to the reduced complexity.

```
 1 #include <stdio.h>
 2 #include <seccomp.h>
 3 #include <unistd.h>
 4 #include <sys/syscall.h>
 5
 6 int main(int argc, char *argv[]) {
 7   int ret;
 8   /* Set default policy: kill everything that
 9       is not explicitly allowed */
10   scmp_filter_ctx ctx;
11   ctx = seccomp_init(SCMP_ACT_KILL);
12   if (ctx == 0) {
13     return 1;
14   }
15   /* Allow write(2). Look at the first argcount
16       (= 1) arguments of the syscall. In this case,
17       the first argument (= 0) is supposed to equal
```

```
     STDOUT_FILENO (which is a makro for 1) */
  ret = seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
      SCMP_SYS(write), 1, SCMP_CMP(0, SCMP_CMP_EQ,
      STDOUT_FILENO));
  if (ret < 0) {
    return 1;
  }
  /* Allow exit_group(2). Ignore arguments to the syscall.
     Now it is possible to do a clean exit by returning at
     the end of main() */
  ret = seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
      SCMP_SYS(exit_group), 0);
  if (ret < 0) {
    return 1;
  }

  /* Load the filter. It is now active. */
  ret = seccomp_load(ctx);
  if (ret < 0) {
    return 1;
  }

  /* This works. */
  write(STDOUT_FILENO, "Hello World!\n", 13);

  /* This doesn't work since STDERR has the file
      descriptor 2 */
  //write(STDERR_FILENO, "Hello World!\n", 13);

  return 0;
}
```

Listing 1.4. Usage of libseccomp

4 Distinction compared to other kernel security features

The Linux kernel provides numerous other features to enhance the security of applications and the system. This chapter enumerates some of them and explains briefly how they relate to seccomp.

4.1 Capabilities

The traditional Unix permission model distinguishes between two types of processes: privileged ones with user ID 0 (usually root, but there can be more users with this user ID) and unprivileged processes with all other user IDs. Unprivileged processes are not able to perform certain operations because their permissions are checked by the kernel, while all actions by a privileged process bypass those checks. There are some tools that need to run privileged for some of their functionality. In order to give regular users access to such binaries, e.g. *ping(8)*, the binary may be marked with the *setuid* permission bit, allowing to be run by a regular user under a privileged context. This coarse grained privilege separation got improved by introducing the concept of *capabilities* [21] in Linux 2.2 that got further improved since then. Lots of processes that required the setuid flag only require a small subset of the available possibilities, ping for example needs only access to networking functionality like for creating ICMP packets. An attacker exploiting such a binary can still use all available privileges to perform other tasks, e.g. performing arbitrary file system accesses. Capabilities aim to split all privileged permissions into groups of permissions that can be granted individually. Ping on a modern Linux system for example has no setuid and the capabilities *cap_net_raw+ep*, meaning that the binary is able to bind to any network port and is able to use raw sockets and packets. The flags *ep* configure detailed behavior that is described in the capabilities manpage [21]. Capabilities are often used in conjunction with seccomp, e.g. for bundling applications into containers and restricting the permissions of those applications as good as possible [22].

4.2 cgroups

Linux has a kernel feature called *control groups*, short *cgroups* [10], that allows to create process groups and enforce and monitor resource limits on them. This makes it possible to prevent Denial of Service style attacks since all resources that may be exhausted can be restricted in such a way that other processes and the system can continue working. This is also a required feature for implementing reliable resource sharing. Cgroups are usually also used for implementing containers or restricting applications that tend to consume lots of resources. Since cgroups address a completely different goal than seccomp, they do not interfere and can be used together.

4.3 namespaces

Namespaces [11] make it possible to create restricted and isolated views on kernel subsystems. This means that a process may be for example started in a different network namespace and can only use a virtual network interface that is internally routed over a VPN connection. This is not detectable for the process and it is impossible to connect to any other network than the provided VPN connection. The man page [11] lists seven namespaces, namely *Cgroup*, *IPC*, *Network*, *Mount*, *PID*, *User*, and *UTS*. By using namespaces, it is possible to provide a process an environment that is to a large extent separated from other processes. The feature is usually used to implement containers. Seccomp can be used additionally to further restrict the processes because namespaces only limit the environment a process is able to see and interact with while seccomp allows to narrow the kernel interface that is exposed.

4.4 chroot

chroot, short for *change root*, allows to change the root directory of a process. The idea is to restrict access to the file system in order to prevent unwanted file access. The classic example where this is desired are web servers[1]. A web server in a chroot can not access files outside of this directory, so vulnerable web applications can not leak or even modify any sensitive host files. On Linux, chroot is rather supposed to prevent errors or misconfiguration. Depending on level of access, it is possible (and even documented in the chroot manpage [12]) for an attacker to escape the chroot. Other systems sometimes provide hardened variants of chroot that try to prevent most kinds of escapes, e.g. FreeBSD's *jails*[13]. Limiting file access can be done via seccomp, too. Nevertheless, chroot does not interfere with seccomp and is easy to implement so using both together is possible.

4.5 LSM

The Linux kernel has a framework named *Linux Security Modules (LSM)* [14] which allows the implementation of *Mandatory Access Control* mechanisms, e.g. *SELinux* and *AppArmor*. Such systems may significantly improve the security of systems by giving operators more flexibility in labeling information and specifying how data is accessed by whom. The downside is that those decisions require a lot of domain knowledge and configuring an effective Mandatory Access Control is a lot of work. The design goals and implementation of LSM is described in [15] while setup and configuration of systems building upon LSM depends on the system used. LSM is only an interface for other applications to build access control, therefore it has different targets and capabilities than seccomp. Usually seccomp can be used together with systems building on LSM, e.g. SELinux.

5 Evaluation

When evaluating security properties of a solution, looking at the history of security relevant bugs can be an interesting indicator for future estimations. There are several CVEs affecting BPF, but taking a closer look reveals that a lot of them focus on BPF directly by injecting malicious bytecode instead of attacking a BPF VM that is already running sane bytecode, e.g. CVE-2016-4557 which allows Denial of Service and privilege escalation by injecting crafted BPF instructions. This is probably not relevant for the seccomp use case because an attacker that has the ability to perform this attack may also modify the sandbox

[1] Although the concept is around for decades, apart from OpenBSD's *httpd* still no web server enforces this today

rule set, effectively removing the seccomp protections without actually abusing a bug. Also, a lot of bugs with CVEs assigned refer to different operating systems: CVE-2017-3196 affects a Windows implementation whereas CVE-2012-3729 was a problem in iOS 6 and despite no CVE assigned, BPF plays an important role in jailbreaking Play Stations [20]. Because most of these attacks abuse BPF as possibility to run code in kernel space by providing crafted bytecode, BPF can be considered a high level target independently from seccomp. Seccomp itself is a far younger project and deals with less complexity, therefore we were only able to find two seccomp related CVEs: CVE-2009-0835 and CVE-2015-2830 which both may allow an application to bypass the seccomp restrictions. Seccomp provides a lot of possibilities to lock down applications and with respect to few bugs that lead to a bypass in the past, using it probably lowers the risk of a successful attack significantly while introducing a low risk of getting exploited because the probably more interesting target for exploitation, BPF, would also exist without seccomp.

6 Applications and higher-level wrappers

This chapter covers applications that make heavy use of seccomp or provide higher-level abstractions to make sandboxing applications using seccomp easier.

6.1 Chromium

Chromium is the open source project behind Google's Chrome browser. When a discussion arose in 2009 if seccomp is actively used, it turned out that Google had already started building a sandbox with the help of seccomp [3]. The Chromium developers were also heavily involved in shaping the feature to how it works today, for example huge patch sets were submitted by them [4].

6.2 Firejail

Firejail is a project that aims to build an easy to use sandbox for existing Linux desktop applications. As such, it has very little dependencies and is able to restrict already existing applications by using seccomp and namespaces. It is shipped with a lot of predefined profiles for well known applications like browser, media players, text editors etc. It has a lot of options and is very flexible while the documentation and the supplied profiles cover a lot of situations, so writing new profiles gets easier by adapting existing profiles.

6.3 NsJail and Kafel

NsJail [25] is a project similar to firejail. It allows to separate processes by using mechanisms like namespaces, chroot, cgroups and seccomp. Isolation of network services, fuzzers and hosting CTF challenges are some of the examples of things that it can be used for. It comes with far less predefined configuration files than firejail, but uses an interesting project called Kafel [26] to write syscall filter policies and compile them to BPF. The Kafel project repository contains some policy examples, for example **sample_basic.policy** as shown in Listing 1.5. It defines a policy called **sample** that allows the *write(2)* syscall if the parameter *myfd* has the value *1* or *2* and the variable *mysize* is either smaller than *4* or the variable *mybuf* is *0*. After the policy definition, it enables the policy and sets the default action for everything not specified by the policy to *KILL*.

```
POLICY sample {
    ALLOW {
        write(myfd, mybuf, mysize) { // filter write call
            (myfd == 1 || myfd == 2) && (mysize < 4
                                        || mybuf == 0)
        }
    }
}

USE sample DEFAULT KILL
```

Listing 1.5. Kafel samples/sample_basic.policy

Even in comparison to using libseccomp, this format is probably better to read, write and understand. For a growing set of different policies this higher level of abstraction makes code and compliance auditing easier.

6.4 mbox

Mbox [27] is a lightweight sandbox built on top of seccomp and a layered file system approach. There is a paper [28] discussing the design decisions. The goal was to create a simple sandbox that is easy to use for non-root users and sandboxes file system access. Mbox creates a new sandbox file system for every execution. All file system accesses of a sandboxed program are forwarded to the sandbox file system. Since this is a regular file system folder, the user can inspect what happened in this directory and use regular Unix tools to operate on files and commit them back to the host system. Mbox uses the possibility to call an external tracer (in this call a modified version of *strace*) on specific syscall events, this way it is possible to rewrite parameters of syscalls that seem to access the regular host file system to another directory, the sandbox file system. This is completely transparent to processes. The project has not seen any commits since May 2014 and may be inactive.

6.5 Docker, LXC and other container frameworks

As already described in Section 4, seccomp is often used in conjunction with container frameworks like Docker [29], LXC [30], rkt [31] and others. These container projects have lots of features and aim to allow large scale application deployments with efficient resource footprints. Traditionally, virtualization used a so-called hypervisor to run virtual operating systems and applications inside. By removing the hypervisor and the guest operating system kernels, containers can be small and generating them is often only a matter of seconds. To address the problem of a shared host kernel, several features of the Linux kernel are used to separate applications from each other and giving the container authors and operators lots of options to control and restrict what containers are able to do.

7 Related Work

This chapters discusses some other frameworks or solutions that try to achieve the same or similar goals. This also covers solutions for other operating systems as seccomp is limited to the Linux kernel.

7.1 OpenBSD: Pledge

The OpenBSD project has a syscall *pledge(2)* [2] that allows to restrict permissions of a process. It is by far not as flexible as seccomp but easier to use. The OpenBSD developers grouped syscalls into *promises*, according to their experiences of syscalls used usually in conjunction. For example, the *rpath* promise contains syscalls for read-only access to a file system. Pledge is easy to implement, leading to some impressive numbers [32] [33]: Pledge is implemented in about 1500 lines of code and using it takes about 3-10 lines of extra code. About 500 programs shipped with OpenBSD's base distribution are shipped with pledge enabled and about 50 from the ports tree support pledge, too. This means that a large part of OpenBSD's system comes with binaries restricted in what they can do, which also demonstrates that performing such a hardening for large parts of the system is absolutely possible while keeping the system functional.

7.2 Systrace

Systrace [34] is a framework to filter syscalls and runs on Linux, but there is also a unmaintained port to macOS and a FreeBSD port is in work. It was also included in earlier releases of OpenBSD. The project is likely dead since there has been no systrace release since 2009. Nevertheless, it first started in 2002 and was status quo in performing syscall filtering for a long time.

7.3 ftrace

During the discussion about the future of seccomp and how to make it more flexible [3], there were arguments for using the *ftrace* framework. It is a rich featured framework for tracing kernel internals. In the discussion about extending seccomp, ftrace already had some of the features implemented, but it's main purpose is to gain insight in the inner workings of the kernel.

8 Conclusion

Seccomp in conjunction with BPF provides a powerful framework for hardening userland programs and implementing sandbox environments. It is shipped with modern Linux kernels and can be used with small to no dependency overhead, depending on the desired level of abstraction. The BPF virtual machine has been in the kernel for a long time, is used by more and more modules and should therefore be considered robust. Apart from OpenBSD, currently no modern desktop or server operating system provides similar native possibilities for syscall filtering regarding the functionality, flexibility and ease of use. Unfortunately, at the time of this writing usage of seccomp seems not very widespread and higher adoption by upstream projects could significantly improve the security of the Linux ecosystem. Chrome proves that it is absolutely possible to sandbox large code bases. The OpenBSD core hardening using *pledge (2)* demonstrates that many small, diverse binaries that are supposed to work with each other can be locked down without users even taking note of it. It may be time to find a way to convince developers that it can be a good idea to implement and enable seccomp in their code running on Linux.

References

1. A thorough introduction to eBPF - LWN.net, https://lwn.net/Articles/740157/. Last accessed 19 April 2018
2. pledge(2), http://man.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2. Last accessed 19 April 2018
3. Seccomp and sandboxing - LWN.net, https://lwn.net/Articles/332974/. Last accessed 24 April 2018
4. Linux Kernel Mailing List - [PATCH v5 2/3] seccomp_filters: system call filtering using BPF, http://lists.openwall.net/linux-kernel/2012/01/27/646. Last accessed 24 April 2018
5. seccomp(2), http://man7.org/linux/man-pages/man2/seccomp.2.html. Last accessed 1 May 2018
6. seccomp(2), http://man7.org/linux/man-pages/man2/exit.2.html. Last accessed 1 May 2018
7. seccomp(2), https://stackoverflow.com/questions/33150281/seccomp-how-to-exit-success/40455896#40455896. Last accessed 1 May 2018
8. System call table x86, https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_32.tbl. Last accessed 1 May 2018
9. System call table x86_64, https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl. Last accessed 1 May 2018
10. cgroups(7), http://man7.org/linux/man-pages/man7/cgroups.7.html. Last accessed 10 May 2018
11. namespaces(7), http://man7.org/linux/man-pages/man7/namespaces.7.html. Last accessed 10 May 2018
12. chroot(2), http://man7.org/linux/man-pages/man2/chroot.2.html. Last accessed 10 May 2018
13. FreeBSD Handbook, Chapter 14. Jails, https://www.freebsd.org/doc/handbook/jails.html. Last accessed 10 May 2018

14. Linux Security Module framework, kernel.org, https://www.kernel.org/doc/Documentation/security/LSM.txt. Last accessed 10 May 2018
15. Wright, C. and Cowan, C. and Morris, J. and Smalley, S. and Kroah-Hartman, G.: "Linux security modules: General security support for the linux kernel" *USENIX Security Symposium.* 2002.
16. McCanne, S. and Van J. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." *USENIX winter. Vol. 93.* 1993.
17. net: filter: rework/optimize internal BPF interpreter's instruction set, git.kernel.org https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8. Last accessed 10 May 2018
18. bpf(2), http://man7.org/linux/man-pages/man2/bpf.2.html. Last accessed 10 May 2018
19. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more https://github.com/iovisor/bcc. Last accessed 12 May 2018
20. PS4 4.55 BPF Race Condition Kernel Exploit Writeup https://github.com/Cryptogenic/Exploit-Writeups/blob/master/FreeBSD/PS4%204.55%20BPF%20Race%20Condition%20Kernel%20Exploit%20Writeup.md. Last accessed 27 May 2018
21. capabilities(7), http://man7.org/linux/man-pages/man7/capabilities.7.html. Last accessed 28 May 2018
22. Red Hat Container Security Guide - Chapter 8. Linux Capabilities and Seccomp, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp. Last accessed 28 May 2018
23. Libseccomp, https://github.com/seccomp/libseccomp. Last accessed 28 May 2018
24. seccomp_rule_add(3), http://man7.org/linux/man-pages/man3/seccomp_rule_add.3.html. Last accessed 28 May 2018
25. NsJail process isolation tool for Linux https://github.com/google/nsjail. Last accessed 29 May 2018
26. Kafel language and library for specifying syscall filtering policies. https://github.com/google/kafel/. Last accessed 29 May 2018
27. A lightweight sandbox tool for non-root users. https://github.com/tsgates/mbox. Last accessed 29 May 2018
28. Kim, Taesoo, and Nickolai Zeldovich. "Practical and Effective Sandboxing for Non-root Users." *USENIX Annual Technical Conference.* 2013.
29. Docker seccomp documentation. https://docs.docker.com/engine/security/seccomp/. Last accessed 29 May 2018
30. LXC seccomp documentation. https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html#lbBD. Last accessed 29 May 2018
31. rkt seccomp documentation. https://coreos.com/rkt/docs/latest/seccomp-guide.html. Last accessed 29 May 2018
32. pledge(), a new mitigation mechanism, Theo De Raadt, OpenBSD project http://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html. Last accessed 29 May 2018
33. Pledge in OpenBSD, Giovanni Bechis https://de.slideshare.net/GiovanniBechis/pledge-in-openbsd. Last accessed 29 May 2018
34. Systrace, http://www.citi.umich.edu/u/provos/systrace/. Last accessed 29 May 2018