Filtering System Calls

in Docker and Kubernetes

Andreas Jaekel, 2019 (K8s 1.16)

whoami

- Working with Linux since the early Jurassic period (1994)
- Head of PaaS Development @ Strato AG
- Responsible for Managed Kubernetes @ IONOS
- Vaguely human

1&1 IONOS

Today's Menu

What is a system call?

Why filter system calls?

Which calls should we filter?

How SysCall filtering works

SysCall filters in Docker

SysCall filters in Kubernetes

Who you gonna call?

WHAT IS A SYSTEM CALL?

Talking to the Kernel

- System Calls are the interface between userland processes and the kernel.
- You can think of them as “the kernel API”.
- Every process uses them. They have to.
- System calls switch the execution domain from “user mode” to “kernel mode” and back.

Pretty please?

A process in Linux receives a slice of memory when it is started. The code is then free to operate on this memory, perform calculations, etc.

For **everything** else, it must ask the kernel for help and permission.

Everything. Really.

- Write to a file? write
- Receive network packets? read
- Get the time of day? gettimeofday
- Create a directory? mkdir
- Send a signal? kill
- Learn its own pid? getpid
- Start a new process? fork

System Calls, and where to find them

There are about 330 system calls in Linux.

The list is here:

/usr/include/asm/unistd_64.h

They all have man pages.

https://syscalls.kernelgrok.com/ (old, though. Linux 2.6.35)

The write() system Call

write()

Used to write data to wherever. (files, network connections, pipes, block devices, etc.)

System Call ID: 1 (in 64-bit mode)

ssize_t write(int fd, const void *buf, size_t count);

Universal Language

All processes must use system calls if they want to get anything done at all.

This is true independent of the programming language they were written in.

write(), in any programming language, will use the write system call internally at some point.

x86-64 Assembler

```
.data
msg: .ascii "Hello World!\n"          # define constant “msg”

.text
.global _start

_start:
   movq $1, %rax                      # use the write() syscall
   movq $1, %rdi                      # write to stdout (file descriptor 1)
   movq $msg, %rsi                    # use string "Hello World!\n"
   movq $13, %rdx                     # write 13 characters
   syscall                            # yield to kernel

   movq $60, %rax                     # use the _exit() syscall
   movq $0, %rdi                      # return code 0
   syscall                            # yield to kernel
```

Plain old C

```
#include <unistd.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
	write(1, "Hello World!\n", 13);
	exit(0);
}
```

Where are the SysCalls?

These function calls actually call the C standard library, not system calls. Let's dig into that.

```
> gcc -Wall -m64 -static -O0 -g -o helloworld \
        helloworld.c
> gdb helloworld
(gdb) disassemble main
```

```
(gdb) disassemble main
main:
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0x4(%rbp)          # Setting up parameters:
mov    %rsi,-0x10(%rbp)
mov    $0xd,%edx                # 13 bytes
mov    $0x493f50,%esi           # address of string
mov    $0x1,%edi                # file descriptor 1
callq  0x410950 <write>         # call libc write()
mov    $0x0,%edi
callq  0x401c70 <exit>
```

GNU libc write() function

```
(gdb) disassemble write
write:
cmpl   $0x0,0x2afc15(%rip)          # if (single_threaded) {
jne    0x41096d <write+29>
mov    $0x1,%eax                    # system call id: 1
syscall                             # yield to kernel
cmp    $0xfffffffffffff001,%rax
jae    0x4139a0
Retq                                # } else {
[…]
```

GoLang

```
package main

import "os"

func main() {
    os.Stdout.Write([]byte("Hello World!\n"))
    os.Exit(0)
}
```

Where are the SysCalls?

These function calls actually call the GoLang standard library, not system calls. Let's dig into that.

> go build helloworld.go

> objdump -d helloworld | less

GoLang disassembly

```
0000000000458570 <main.main>:
[...]
mov    %rcx,0x8(%rsp)
mov    %rdx,0x10(%rsp)
mov    %rbx,0x18(%rsp)
callq  457220 <os.(*File).Write>
[...]
```

GoLang disassembly

```
0000000000457220 <os.(*File).Write>:
[…]
callq  457c60 <os.(*File).write>
[…]
```

GoLang disassembly

```
0000000000457c60 <os.(*File).write>:
[…]
callq  456bd0 <internal/poll.(*FD).Write>
[…]
```

GoLang disassembly

```
0000000000456bd0 <internal/poll.(*FD).Write>:
[...]
callq  454e80 <syscall.Write>
[...]
```

GoLang disassembly

```
0000000000454e80 <syscall.Write>:
[...]
callq  455330 <syscall.write>
[...]
```

GoLang disassembly

```
0000000000455330 <syscall.write>:
[…]
movq   $0x1,(%rsp)                    # systemcall id: 1
mov    0x58(%rsp),%rdx
mov    %rdx,0x8(%rsp)
mov    %rcx,0x10(%rsp)
mov    %rax,0x18(%rsp)
callq  455790 <syscall.Syscall>
[…]
```

GoLang disassembly

```
0000000000455790 <syscall.Syscall>:
[…]
mov    0x8(%rsp),%rax
syscall                              # yield to kernel
[…]
```

Recap

Assembler

_start

syscall

Recap

C

main()

write()

syscall

Recap

Golang

main.main()
os.(*File).Write()
os.(*File).write()
internal/poll.(*FD).Write()
syscall.Write()
syscall.write()
syscall.Syscall()
syscall

Java

Just kidding.

The point is…

No matter what programming language you use, internally they will all have to use system calls to get anything done.

Which makes system calls a perfect place to limit what processes can do.

Snakes. Why did it have to be snakes?

WHY FILTER SYSTEM CALLS?

Why filter SysCalls?

Only one simple reason:

To prevent programs and containers from doing anything we do not want them to do.

Attack Vectors

1. Backdoors in Docker upstream images
2. Exploitable bugs in our application
3. Vulnerable system calls in our Linux kernel

Example: Nginx non-Calls

In order to limit what the container can do, we disable system calls that we're sure Nginx will never need. Such as:

- kill
- ptrace
- swapoff
- truncate
- setxattr
- capset
- reboot
- mkdir
- link
- creat
- mount
- setuid
- rename
- rmdir
- umount
- chroot
- symlink
- swapon
- sethostname
- setpriority
- init_module
- delete_module
- quotactl
- etc.

Obvious Advantages

- Nginx does not need these system calls
- If we block them Nginx will be fine
- Software that is not supposed to be there can not use these calls
- This makes us safer. Reduces attack surface.

Nuke the site from orbit. It’s the only way to be sure.

WHICH CALLS SHOULD WE FILTER?

Which System Calls will be used?

1. Read the source. All of it. All libraries, too!
2. Educated guessing.
3. Analyzing the binaries.
4. Tracing/auditing system calls at runtime.
5. Trial and error.

Picking Filters – reading sources

- obviously infeasible
- never-ending dependency chains
- code analysis tools? Nope: eval(), pre-processor macros, obfuscation, etc.
- unfortunately, the only way to be almost sure there's no evil code

Picking Filters – educated guessing

- requires knowledge of software design and system calls
- will likely either filter too much or too little, or both
- will actually stop backdoors in the images that need additional calls
- it’s better than nothing

Picking Filters – binary analysis

- only viable for static binaries
- dynamic run-time linking -> ALL the syscalls!
- might be difficult. (remember GoLang's general-purpose syscall.Syscall function?)
- will not stop backdoors in the image, but might stop malicious code downloaded at runtime.

Picking Filters – call tracing

- test-run the image, trace all calls and make a list of all system calls it uses.
- easy to do.
- but make sure to trace all use cases.
- will not stop backdoors in the image, but might stop malicious code downloaded at runtime.

Only Tracing is feasible

- use call tracing
- possibly during your CI pipeline or unit tests
- add Docker-required syscalls if necessary

By the way: Docker already has a good default filter – but it must necessarily allow for all kinds of applications to work.

Tracing System Calls

Linux offers “strace”.

Will output all system calls used by the traced thread, and optionally all new threads.

Great tool for debugging and trouble shooting.

Will also show call parameters.

strace Example

```
> strace ./helloworld
execve("./helloworld", ["./helloworld"], [/* 29 vars */]) = 0
uname({sysname="Linux", nodename="goto", ...}) = 0
brk(NULL)                               = 0x1dcf000
brk(0x1dd01c0)                          = 0x1dd01c0
arch_prctl(ARCH_SET_FS, 0x1dcf880)      = 0
brk(0x1df11c0)                          = 0x1df11c0
brk(0x1df2000)                          = 0x1df2000
write(1, "Hello World!\n", 13Hello World!
)         = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Strace Example

strace also has a “counter mode”:

```
> strace -c -S name ./helloworld
Hello World!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         4           brk
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         1           write
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                     8           total
```

Recap

- Writing a good filter list is hard.
- Filter too much and the app stops working.
- Filtering too little leaves more room for bad guys.
- strace can help find all required system calls.
- Use educated guesses to double-check.

The Voight-Kampff Test.

HOW SYSCALL FILTERING WORKS

The Short Version

- Filters are implemented as small programs.
- There is a system call to load and apply a filter program to a running process
- Once the filter is applied, the system calls are being filtered.

Filter Programs

Linux has the ability to execute small state machines before any system call.

These programs must be delivered as compiled eBPF bytecode binaries. (“extended Berkeley Packet Filter”)

They can be loaded into the kernel with the bpf() system call.

Multi-purpose Tool eBPF

eBPF can be used for all kinds of things:

- Performance Measurements
- Tracing
- Debugging
- Filtering network packets
- etc.

eBPF is hard

Writing BPF programs is complex. It’s not unlike writing assembly.

But we don’t want to learn a new programming language right now – we just want to filter system calls.

Introducing: Seccomp BPF

Seccomp was created by Google in 2005.

It offered a “strict mode” to only allow read(), write(), sigreturn() and exit().

In 2012 it learned to use eBPF internally, and now also offers filtering of individual system calls.

Seccomp hides the complexity of eBPF from the user.

Example: using Seccomp in C

```
int
main(int argc, char *argv[])
{
  scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
  seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(getpid), 0);
  seccomp_load(ctx);

  pid_t pid = getpid();
  /* never reached: process killed */
  return 0;
}
```

Filter by SysCall Paramters

Seccomp can filter based on the parameters:

```
unsigned char buf[BUF_SIZE];
int fd = open(“data.raw", 0);
int rc = seccomp_rule_add(
            ctx,
            SCMP_ACT_ALLOW,
            SCMP_SYS(read), 3,
            SCMP_A0(SCMP_CMP_EQ, fd),
            SCMP_A1(SCMP_CMP_EQ, (scmp_datum_t)buf),
            SCMP_A2(SCMP_CMP_LE, BUF_SIZE));
```

Filtering by Parameters

This can be useful for a lot of reasons. Some examples:

- force read-only system calls
- limit reads and writes to STDOUT and STDIN
- limit setuid() to specific UIDs
- forbid sending signals other than SIGHUP
- prevent setting super-generous file permission
- etc.

Limitations to Parameter Filtering

- Only “pass by value” parameters.
- Can’t look into userland memory.
- That means no peeking into strings or structures.
- Example: can’t limit open() to certain filenames.

Seccomp BPF - not just for filtering

Seccomp BPF can not just allow or block system calls.

- can pretend the syscall happened, but it didn't
- can return fake results and error numbers
- can trigger breakpoints (trace points)

Ergo, Seccomp BPF is also good for testing, error injection and debugging.

We got Country AND Western!

SYSCALL FILTERS IN DOCKER

Seccomp and Docker

Seccomp support was added to Docker in v1.10.

44 syscalls are blocked by default. They include reboot(), a few obsolete ones, and exotic ones that had exploits. Some are only allowed if the kernel is fresh enough.

Undesired syscalls will fail, but the program isn't killed.

Custom Filters in Docker

Custom filters are expressed as JSON files.

The JSON structure enables a subset of Seccomp's abilities, much as Seccomp enables a subset of eBPFs abilities.

When writing a custom filter, it is recommended to start with the default filter and adjust it as needed.

Example Docker Seccomp Profile

```
{
        "defaultAction": "SCMP_ACT_ERRNO",
        "syscalls": [
                {
                        "names": [
                                "accept",
                                "access",
                                ...
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": [],
                        "comment": "",
                        "includes": {},
                        "excludes": {}
                },
                {
                        "names": [
                                "ptrace"
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": null,
                        "comment": "",
                        "includes": {
                                "minKernel": "4.8"
                        },
                        "excludes": {}
                }
        ]
}
```

default action: fail with error.

allow all these system calls, regardless of their parameters.

allow ptrace(), but only on kernels newer or equal linux-4.8

Loading a Filter Set in Docker

The filter JSON file (Docker calls it a “seccomp profile”) can be given as a command line parameter:

```
# docker run -ti --rm --security-opt  \
        seccomp:custom_filter.json alpine /bin/sh
```

Docker Seccomp Caveats

- Any seccomp profile given will replace the default one, not extend it.
- The filter will apply to the whole container.
- Additional syscalls are required by Docker to bootstrap containers. (18 syscalls)

The Master Control Program

SYSCALL FILTERS IN KUBERNETES

SysCall Filters in Kubernetes

- Added in Kubernetes 1.3 (2016)
- Supported by most runtimes, not just Docker.
- Seccomp profiles are still an alpha feature.
- Seccomp profiles apply to the entire pod, not just to any single container.
- The default is not to allow custom profiles.

Prerequisites

- enable Pod Security Policies in the K8s cluster
- define a pod security policy that allows seccomp profiles to be used
- create a RoleBinding so that pods may use this policy

Activating PodSecurityPolicies

- Add at least one permissive policy before activating the admission controller.
- Also, create at least one matching role and a role binding for the kube-system namespace.
  Otherwise K8s will not be able to start any pods.
  (including system pods such as kube-api, etc.)

Activating PodSecurityPolicies

Then, add PodSecurityPolicy to the list of enabled admission controllers:

```
kube-apiserver \
        --enable-admission-plugins= \
                PodSecurityPolicy,LimitRanger ...
```

Provide Seccomp Profiles

- Write the profiles. Format depends on your runtime.
- Place them on the worker nodes.
- Where?

  --kubelet --seccomp-profile-root=
- Default:

  /var/lib/kubelet/seccomp

Applying a Seccomp Filter to a Pod

Add annotations to the pod (template):

```
[…]
metadata:
    labels:
      app: problemsolver
annotations:
    kubernetes.io/psp: privileged
    seccomp.security.alpha.kubernetes.io/pod: localhost/custom-profile.json
[…]
```

Example Files

Download the example files to get a quick start:

git clone \
https://github.com/ionos-enterprise/K8s-seccomp-demo

(not for production – too permissive)

There Can Only Be One.

DEMO

Wax on. Wax off.

CLOSING REMARKS

Is it worth it?

For most people… probably not.

- Docker defaults are quite good.
- Too strict? Your application misbehaves.
- Too generous? Sandbox not perfect. (ok?)
- Lots of effort finding the right set of syscalls.
- Might be time consuming to keep up with changes in the application code.

When should I use this?

It comes down to making a cost-benefit analysis.

If you need super secure sandboxes (fin-tech? defense contractor? coffee production?) you would probably use VMs anyway, not containers.

When should I use this?

- can be a quick win if you know your application well
- or if writing filters is easy for any other reason
- very much worth the effort if you are a container hoster
- or if you offer server-less runtimes in containers
- or if you offer other kinds of containers that can run arbitrary user code. (WordPress?)
- if you need all the security you can get

Take Away

- System calls are the Kernel API to user land.
- All Linux software uses system calls. It has to.
- System calls can be limited with BPF filters.
- BPF is powerful, yet complex. Can do much more.
- Seccomp makes this easier. Less powerful, though.
- Docker and Kubernetes support this. We can apply custom filters written in JSON.
- There’s effort involved, but it’s good to know the option exists - because sometimes it’s worth it.

Thank you!

Andreas Jaekel

1&1 IONOS

Excellent!

BONUS SLIDES

List of System Calls Docker requires

No matter what's running in the image, these are always needed:

capget	prctl	getdents64
capset	setgroups	newfstatat
chdir	setuid	rt_sigaction
chown	futex	rt_sigreturn
lstat	exit	setgid
openat	fstat	write

eBPF: limit open() to read-only

```
BPF_STMT(BPF_LD | BPF_W | BPF_ABS ,(offsetof(structseccomp_data , arch))),
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K , AUDIT_ARCH_X86_64 , 1, 0),
BPF_STMT(BPF_RET | BPF_K , SECCOMP_RET_KILL),
BPF_STMT(BPF_LD | BPF_W | BPF_ABS , (offsetof(structseccomp_data , nr))),
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K , __NR_open , 1, 0),
BPF_STMT(BPF_RET | BPF_K , SECCOMP_RET_ALLOW),
BPF_STMT(BPF_LD | BPF_W | BPF_ABS , (offsetof(structseccomp_data , args [1]))),
BPF_JUMP(BPF_JMP | BPF_JSET | BPF_K , O_CREAT , 0, 1),
BPF_STMT(BPF_RET | BPF_K , SECCOMP_RET_KILL),
BPF_JUMP(BPF_JMP | BPF_JSET | BPF_K , O_WRONLY | O_RDWR , 0, 1),
BPF_STMT(BPF_RET | BPF_K ,SECCOMP_RET_ERRNO |(ENOTSUP &
SECCOMP_RET_DATA )),
BPF_STMT(BPF_RET | BPF_K , SECCOMP_RET_ALLOW)
```

Links

- “Hello World” analysis: http://osteras.info/personal/2013/10/11/hello-world-analysis.html
- Pod Security Policy Documenation: https://kubernetes.io/docs/concepts/policy/pod-security-policy/

The Two Minute Minikube Version

```
# Download demo files
git clone https://github.com/ionos-enterprise/K8s-seccomp-demo
cd K8s-seccomp-demo

# Provide default policy and roles for minkube:
mkdir -p ~/.minikube/files/etc/kubernets/addons
cp psp.yaml ~/.minikube/files/etc/kubernets/addons

# Provide custom seccomp profile:
mkdir -p ~/.minikube/files/var/lib/kubelet/seccomp
cp limited-seccomp-profile.yaml ~/.minikube/files/var/lib/kubelet/seccomp

# Start minikube and deploy demo app
minikube start --extra-config=apiserver.enable-admission-plugins="PodSecurityPolicy“
kubectl apply -f nginx-hello-deployment.yaml
```

The Two Minute Minikube Version

```
# Verify that nginx is working
kubectl port-forward $POD 80
curl localhost:80

# Verify that some system calls are disabled: (unlink)
kubectl exec -ti $POD /bin/sh
~ touch foo
~ rm foo
rm: can't remove 'foo': Operation not permitted

# Verify that some system calls are disabled: (mknod)
~ mknod bar c 10 10
mknod: bar: Operation not permitted
```

Possible Demos

- Use strace to make a syscall list for the nginx hello world container
- Use strace to make a syscall list for /bin/sh
- Start a restricted /bin/sh container that can’t fork
- Deploy a restricted nginx-hello app in K8s

strace - make a syscall list for /bin/sh

- docker run -ti --rm alpine:latest /bin/sh
- Outside: find the /bin/sh process
- strace -f -c -p $pid
- Inside: exec /bin/sh
- Take the list and make it into a Docker seccomp profile, based on the default. Add the required syscalls for Docker.
- Restart the container with the profile active.
- Adding fork() allows things like “ls”
- Adding utimensat() allows “touch test”

Demo: nginx-hello in K8s

1. Set up minkube with our demo files: https://github.com/ionos-enterprise/K8s-seccomp-demo
2. kubectl apply –f nginx-hello-deployment.yaml
3. Show that nginx is running (kubectl port-forward)
4. Open a shell: kubectl exec -ti $POD /bin/sh
5. Demonstrate limitation: touch foo; rm foo

Coma

Without system calls, a process can not be useful.
(unless you just want to heat up the CPU)

It can’t get any data, and it can’t display or persist any results anywhere, or change anything.