Kernel Hacker Stuff - Seccomp

Jake Drahos

February 3, 2022

Overview

The SECure COMPuting kernel facility

Seccomp is a kernel facility providing a one-way transition into “secure computing” mode, restricting which parts of the kernel API are accessible to the process.[1]

[1] *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

SECCOMP IS NOT A SANDBOX

(this disclaimer itself is paraphrased from the official documentation)

Seccomp, on its own, is neither a security feature nor a sandbox!

SECCOMP IS NOT A SANDBOX

(this disclaimer itself is paraphrased from the official documentation)

Seccomp, on its own, is neither a security feature nor a sandbox!

However, it can be (and is) used to implement those things.

SECCOMP IS NOT A SANDBOX

(this disclaimer itself is paraphrased from the official documentation)

Seccomp, on its own, is neither a security feature nor a sandbox!

However, it can be (and is) used to implement those things.

While it isn't something you just "turn on and get security", it is important (or at least interesting) to understand the kernel facilities that support useful security features).

Operating Systems 101

Processor Execution Modes

- Modern (post-1980s) processors are designed to support the concept of “process isolation” via multiple privilege levels
- User/process privilege level sees the machine as only its abstract ISA
    - “Privilege level” in an ISA is a different concept that an OS user’s privilege level
- Any interaction with the outside world must be done by the kernel on behalf of the process!

The syscall instruction

ISAs provide a system call instruction (`int 80h`, `syscall`, `ecall`, `t 0x6d`, etc.)

The syscall instruction

ISAs provide a system call instruction (`int 80h`, `syscall`, `ecall`, `t 0x6d`, etc.)

What the processor does when a syscall instruction executes:

1. Save execution state (PC, SP, GP registers, condition...)
2. Switch to kernel-mode memory map
3. Increase privilege level
4. Begin executing instructions at a configured address

The address where kernel-mode execution begins is the syscall entry handler

System Calls as Function Calls

A syscall API can be standardized just like any other library API.

System Calls as Function Calls

A syscall API can be standardized just like any other library API.

Processes can execute a syscall by:

1. Setting up any buffers (allocate/copy memory)
2. Place arguments (pointers, flags, sizes...) where the ABI dictates
3. Execute the syscall instruction

Kernel looks at arguments, info about the process (user, OS permissions...) and decides if/how to fulfill the request.

The Linux Syscall Interface

Syscalls ultimately provide the only* mechanism by which processes can do anything interesting.

Syscalls are documented in man(2), plus kernel documentation for ABI details

Syscalls ultimately provide the only* mechanism by which processes can do anything interesting.

Syscalls are documented in man(2), plus kernel documentation for ABI details

* getting creative with memory mapping (and evil x86 in/out instructions) can allow direct access to IO (or shared memory), but require syscalls to set up.

Traditional Unix IO

Simple, standard file operations

- open()
- close()
- read()
- write()

Traditional Unix IO

Simple, standard file operations

- open()
- close()
- read()
- write()

(actually the *at versions tend to be used by glibc because reasons)

Process Management

Unix process lifecycle managed by syscalls

- `fork()`
- `execve()`
- `wait()`

Memory map manipulation

Some syscalls allow a process to manipulate its own memory mapping

- `sbrk()`
- `mmap()`
- `munmap(`

Ugly and evil syscalls

A few nasty ones:

- `ioctl()` **HERE THERE BE DRAGONS**
- `socket()`
- `send()`
- `recv()`

And approximately 300 more.

“Non-syscall” interfaces

Better API design than `ioctl` through a few different interfaces:

- `procfs`
- `sysfs`

“Non-syscall” interfaces

Better API design than `ioctl` through a few different interfaces:

- `procfs`
- `sysfs`

But these all eventually rely on `read()`/`write()`.

“Non-syscall” interfaces

Better API design than `ioctl` through a few different interfaces:

- `procfs`
- `sysfs`

But these all eventually rely on `read()`/`write()`.

Even direct accesses to mapped IO space require a `mmap()` syscall to set up.

Kernel Attack Surface

Syscalls represent the kernel's attack surface from userspace (IO attack surface eg. ping-of-death is a different story)

A naughty process can't exploit kernel bugs in syscalls it can't use.

Userspace defense-in-depth

Restricting syscalls can also protect a process from itself.

Userspace defense-in-depth

Restricting syscalls can also protect a process from itself.

Classic RCE:

1. Buffer overflow
2. ROP or shellcode
3. `execve(‘‘/bin/sh’’, null)`
4. Profit!

Userspace defense-in-depth

Restricting syscalls can also protect a process from itself.

Classic RCE:

1. Buffer overflow
2. ROP or shellcode
3. `execve(‘‘/bin/sh’’, null)`
4. Profit!

Good luck with that if you can’t use the `execve()` syscall!

Sandboxing

SECCOMP IS NOT A SANDBOX

Sandboxing

SECCOMP IS NOT A SANDBOX

Intercepting or filtering syscalls can harden a sandbox:

Sandboxing

SECCOMP IS NOT A SANDBOX

Intercepting or filtering syscalls can harden a sandbox:

- Preventing opening new files
- Disallow execution of other executables
- Disallowing access to devices
- Aggressively whitelisting only `read()`s and `write()`s
- ...

Seccomp

Enter: seccomp

`seccomp` is the kernel feature that allows syscall filtering

Matching rules and behavior (perform as normal, `EINVAL`, send a signal) set during the transition to seccomp mode - a one-way trip.

Technically happens over a few different `prctl` syscalls to handle the processes privilege level and set up prerequisites to avoid breakout.

SECCOMP_SET_MODE_STRICT

Seccomp initially supported only “strict” mode.

Simple, well-defined behavior:

- `read()`
- `write()`
- `_exit()`

SECCOMP_SET_MODE_STRICT

Seccomp initially supported only “strict” mode.

Simple, well-defined behavior:

- `read()`
- `write()`
- `_exit()`

Nothing else allowed!

SECCOMP_SET_MODE_STRICT

Seccomp initially supported only “strict” mode.

Simple, well-defined behavior:

- `read()`
- `write()`
- `_exit()`

Nothing else allowed! In addition to inflexibility, this totally breaks libc (hidden `sbrk()`s, `*at` syscalls, ...).

SECCOMP_SET_MODE_FILTER

This is where all the cool stuff lives!

- Flexibility with which calls to filter.
- Allow, crash, error, or signal

[2] Chromium. *Linux Sandboxing*. https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md.

SECCOMP_SET_MODE_FILTER

This is where all the cool stuff lives!

- Flexibility with which calls to filter.
- Allow, crash, error, or signal

Signalling offers interesting options to emulate IO functionality for a sandbox[2].

[2] Chromium, *Linux Sandboxing*.

BPF intro

The BSD Packet Filter[3] was designed for matching on binary data in network packets.

[3] Steven McCanne and Van Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. Tech. rep. Lawrence Berkely Laboratory, 1992.

BPF intro

The BSD Packet Filter[3] was designed for matching on binary data in network packets.

Classic BPF operates as a simple accumulator-based virtual machine

- Accumulator and index(x) registers
- Scratch memory
- Read-only data memory (the packet)

[3] McCanne and Jacobson, *The BSD Packet Filter: A New Architecture for User-level Packet Capture*.

Funny note:

SCO pointed to BPF as proof that Linux stole code from them...

[4] Bruce Perens and Linus Torvalds. *Analysis of SCO's Las Vegas Slide Show*. https://web.archive.org/web/20090217185248/http://perens.com/Articles/SCO/SCOSlideShow.html.

Funny note:

SCO pointed to BPF as proof that Linux stole code from them... despite SCO never owning BSD (and therefore the original BPF code) in the first place[4].

[4] Perens and Torvalds, *Analysis of SCO's Las Vegas Slide Show*.

BPF instruction encoding

Simple instruction encoding (two words per instruction):

<table>
<tr><td>opcode:16</td><td>jt:8</td><td>jf:8</td></tr>
<tr><td colspan="3">k:32</td></tr>
</table>

Not all opcodes use jt, jf, and k

jt and jf encode *unsigned* jump distances for conditional jumps

k encodes a constant

Example opcodes

- `ld[b|h]` (load from data to accumulator)
- `st` (store to scratch memory)
- `jne`, `jgt`, etc. (conditional jumps)
- `add`, `mul`, `or`, etc. (arithmetic and bitwise)

Address modes

Non-exhaustive list.

- Constant (encoded `k` in the instruction)
- `x`
- `[k]` or `[x+k]` (data)

Example

```
ldb [14]
and #0xf
lsh #2
tax
ldh [x+16]
jeq #80, 0, 1
ret 1
ret 0
```

But how does it apply to seccomp!

BPF programs are used to implement the logic for syscall filtering with seccomp.

But how does it apply to seccomp!

BPF programs are used to implement the logic for syscall filtering with seccomp.

The implementation is pretty basic:

- "Packet Data" is a `struct seccomp_data`
- BPF program return value determines behavior (allow/deny/EINVAL)
- Flexible logic:
  - Masking operations to check flags in arguments
- *No* ability to deference pointers (avoid race conditions)

But how does it apply to seccomp!

BPF programs are used to implement the logic for syscall filtering with seccomp.

The implementation is pretty basic:

- "Packet Data" is a `struct seccomp_data`
- BPF program return value determines behavior (allow/deny/EINVAL)
- Flexible logic:
    - Masking operations to check flags in arguments
- *No* ability to deference pointers (avoid race conditions)

It is admittedly a bit overkill (eBPF is a story for another day).

struct seccomp_data

```
#include <seccomp.h>
struct seccomp_data {
    int     nr,
    _u32    arch,
    _u64    instruction_pointer,
    _u64    args[6]
};
```

BPF Programs in code

Classic BPF is easy enough to hand-write in C.

Macros and structs provided to make it easy to express a program:

```c
struct sock_filter filter[] = {
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, FILTER_ARCH, 1, 0),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (EPERM & SECCOMP_RET_DATA)),
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_read, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_write, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_writev, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_open, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (EPERM & SECCOMP_RET_DATA)),
};
```

Applications

- Chromium leverages seccomp for attack surface reduction
  - also putting flash in time-out
- Non-root sandboxing[5]
- Theoretical: mobile-style app sandboxing with native code
  - Userspace platform API calls into “trusted” regions to do syscalls
  - Sort of how VDSO and bits of NT already work
  - ROP exists...
- QEMU (attack surface reduction)
- Containers (magical christmas land)

[5] Taesoo Kim and Nickolai Zeldovich. *Practical and effective sandboxing for non-root users*. Tech. rep.

- Chromium. *Linux Sandboxing*. https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md.
- Kim, Taesoo and Nickolai Zeldovich. *Practical and effective sandboxing for non-root users*. Tech. rep.
- McCanne, Steven and Van Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. Tech. rep. Lawrence Berkely Laboratory, 1992.
- Perens, Bruce and Linus Torvalds. *Analysis of SCO's Las Vegas Slide Show*. https://web.archive.org/web/20090217185248/http://perens.com/Articles/SCO/SCOSlideShow.html.
- *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.