Revisiting eBPF Seccomp Filters

Jinghao Jia
University of Illinois at Urbana-Champaign
LPC ’22 (Networking & BPF Track)
Seccomp

• The syscall interposition mechanism for Linux
  • Restricts the set of syscalls for an application
  • First implemented by Andrea Arcangeli and merged in kernel 2.6.12
Seccomp

• The syscall interposition mechanism for Linux
  • Restricts the set of syscalls for an application
  • First implemented by Andrea Arcangeli and merged in kernel 2.6.12

• Two modes
Seccomp

• The syscall interposition mechanism for Linux
  • Restricts the set of syscalls for an application
  • First implemented by Andrea Arcangeli and merged in kernel 2.6.12
• Two modes
  • Restricted mode: only allows read(), write(), exit(), and sigreturn()
    • Kills the process if other syscalls are used
Seccomp

- The syscall interposition mechanism for Linux
  - Restricts the set of syscalls for an application
  - First implemented by Andrea Arcangeli and merged in kernel 2.6.12
- Two modes
  - Restricted mode: only allows read(), write(), exit(), and sigreturn()
    - Kills the process if other syscalls are used
  - Filter mode: allows custom policies implemented in `cBPF` (classic BPF)
    - Custom set of allowed/disallowed syscalls and non-pointer arguments
    - Custom actions (e.g., allow, return error code, kill process)
cBPF is simple

- Small, RISC-like instruction set
- Only allows forward jumps
- Limited to 4096 instructions in length
cBPF is simple

- Small, RISC-like instruction set
- Only allows forward jumps
- Limited to 4096 instructions in length
Seccomp-cBPF is widely used
Seccomp-cBPF is widely used

- Virtualization/Containers
  - Docker, Rkt, gVisor, Kubernetes, etc
Seccomp-cBPF is widely used

- Virtualization/Containers
  - Docker, Rkt, gVisor, Kubernetes, etc

- Sandboxing
  - Systemd, Android
Seccomp-cBPF is widely used

• Virtualization/Containers
  • Docker, Rkt, gVisor, Kubernetes, etc

• Sandboxing
  • Systemd, Android

• Various other applications
  • OpenSSH, Tor, Firefox, …
Limitation of cBPF

Limitation of cBPF

• **Stateless**: only support **static** allow/deny lists

Limitation of cBPF

- **Stateless**: only support static allow/deny lists
- **Size limit**: complex policies cannot be encoded in one filter [1]
Limitation of cBPF

- **Stateless**: only support **static** allow/deny lists
- **Size limit**: complex policies cannot be encoded in one filter [1]
- **Simple instruction set**: cannot use advanced utilities
Limitation of cBPF

- **Stateless**: only support **static** allow/deny lists
- **Size limit**: complex policies cannot be encoded in one filter [1]
- **Simple instruction set**: cannot use advanced utilities
- **Programmability**: cannot use modern eBPF tools

Limitation of cBPF

- **Stateless**: only support static allow/deny lists
- **Size limit**: complex policies cannot be encoded in one filter [1]
- **Simple instruction set**: cannot use advanced utilities
- **Programmability**: cannot use modern eBPF tools
  - Requires writing cBPF instructions by hand (mitigated by libseccomp)

Limitation of cBPF

- **Stateless**: only support *static* allow/deny lists
- **Size limit**: complex policies cannot be encoded in one filter [1]
- **Simple instruction set**: cannot use advanced utilities
- **Programmability**: cannot use modern eBPF tools
  - Requires writing cBPF instructions by hand (mitigated by libseccomp)

cBPF cannot express complex syscall policies, e.g., state-based filtering

Revisiting eBPF Seccomp filters to improve *programmability* and *extensibility* of system call security.
Advances of eBPF

A new level of programmability
Advances of eBPF

A new level of programmability

- Richer instruction set and more flexible control flow
Advances of eBPF

A new level of programmability

• Richer instruction set and more flexible control flow
• eBPF map structures for efficient storage
Advances of eBPF

A new level of programmability

• **Richer instruction set and more flexible control flow**
• **eBPF map structures** for efficient storage
• **Helper interface** for rich interaction with kernel
Advances of eBPF

A new level of programmability

- Richer instruction set and more flexible control flow
- eBPF map structures for efficient storage
- Helper interface for rich interaction with kernel
- Toolchain support: write C, compile to eBPF
Implications on Seccomp
Implications on Seccomp

- Advanced checking logic from new instruction set
Implications on Seccomp

- Advanced checking logic from new instruction set
- Stateful filters via map storage
Implications on Seccomp

- Advanced checking logic from new instruction set
- Stateful filters via map storage
- Access to more kernel context using kernel helper functions
Implications on Seccomp

• Advanced checking logic from new instruction set
• Stateful filters via map storage
• Access to more kernel context using kernel helper functions
• Mature toolchains
Previous efforts

• Seccomp-eBPF patch from Dhillon (Feb. 2018)
  • https://lwn.net/Articles/747229/
  + A new Seccomp program type for eBPF
  + A new eBPF flag in Seccomp with code that allocates eBPF filters
    - No eBPF map support nor security model
    - Only supports 4 helpers

• Seccomp-eBPF patch from Hromatka (Feb. 2018)
  • https://groups.google.com/g/libseccomp/c/pX6QkVFoF74?pli=1
  + A new eBPF filter mode in Seccomp that loads and allocates the filter.
    - Incomplete -- no Seccomp-eBPF program type/whitelisted helpers
Some common opinions
Some common opinions

• “What's the reason for adding eBPF support? seccomp shouldn't need it, and it only makes the code more complex. I'd rather stick with cBPF until we have an overwhelmingly good reason to use eBPF as a "native" seccomp filter language.”

Lacking use cases
Some common opinions

• “What's the reason for adding eBPF support? seccomp shouldn't need it, and it only makes the code more complex. I'd rather stick with cBPF until we have an overwhelmingly good reason to use eBPF as a "native" seccomp filter language.”

• This is the blocker as far as I'm concerned: there is no story for unprivileged eBPF. And even IF there was a story there, I find the rate of security-related flaws in eBPF to be way too high for a sandboxing primitive to depend on...I just can't bring myself to accept that level of risk for seccomp.
Many use cases of Seccomp-eBPF

- Syscall count limiting
- Enhanced temporal syscall specialization
- Syscall-flow-integrity protection
- Checking syscall sequence for intrusion detection systems
- Accelerating security checks
- Many others!
Syscall count limiting

Real world example in runc
Syscall count limiting

Real world example in runc

```go
if l.cfg.Config.Seccomp != nil &&
  !l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
}
```

Sets up Seccomp filters
Syscall count limiting

Real world example in runc

```go
if l.cfg.Config.Seccomp != nil &&
    !l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
}
finalizeNamespace(...)
...
if unix.Getppid() != l.parentPid { ...
...
unix.Write(fd, []byte("0"))
```

Sets up Seccomp filters
namespace & privilege setup, etc
Syscall count limiting

Real world example in runc

```go
if l.cfg.Config.Seccomp != nil &&
    !l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
}
finalizeNamespace(...)
...
if unix.Getppid() != l.parentPid { ... }
...
unix.Write(fd, []byte("0"))
system.Exec(name, l.cfg.Args[0:], os.Env())```

Sets up Seccomp filters
namespace & privilege setup, etc
exec into target application
Syscall count limiting

Real world example in runc

```go
if l.cfg.Config.Seccomp != nil &&
!l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
}
finalizeNamespace(...)
...
if unix.Getppid() != l.parentPid { ... }
...
unix.Write(fd, []byte("0"))
system.Exec(name, l.cfg.Args[0:], os.Env())
```

seccomp w/o NO_NEW_PRIVS is privileged!
Needed before dropping privileges
Syscall count limiting

Real world example in runc

if l.cfg.Config.Seccomp != nil &&
  !l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
  }

class finalizeNamespace(...) {
  ...
  if unix.Getppid() != l.parentPid {
    ...
  }

  unix.Write(fd, []byte("0"))
  system.Exec(name, l.cfg.Args[0:], os.Env())

  syscalls for later setups need to be allowed
  e.g. execve, prctl, capset
Syscall count limiting

Real world example in runc

```go
if l.cfg.Config.Seccomp != nil &&
!l.cfg.NoNewPrivileges {
    seccomp.InitSeccomp(...)
}
finalizeNamespace(...)
...
if unix.Getppid() != l.parentPid { ... }
...
unix.Write(fd, []byte("0"))
system.Exec(name, l.cfg.Args[0:], os.Env())
```

Syscalls for later setups need to be allowed e.g. execve, prctl, capset

cBPF filters cannot block setup syscalls during application run.
Syscall count limiting

- Solution: Implement a state-based policy in eBPF
Syscall count limiting

• Solution: Implement a state-based policy in eBPF

```c
unsigned int no_exec;
SEC("seccomp")
int exec_once(struct seccomp_data *ctx)
{
    if (ctx->nr == __NR_execve) {
        if (!no_exec) {
            no_exec = 1;
            return SECCOMP_RET_ALLOW;
        }
        return SECCOMP_RET_ERRNO | EPERM;
    }
    return SECCOMP_RET_ALLOW;
}
```
Syscall count limiting

- Solution: Implement a state-based policy in eBPF

```c
unsigned int no_exec;
SEC("seccomp")
int exec_once(struct seccomp_data *ctx)
{
    if (ctx->nr == __NR_execve) {
        if (!no_exec) {
            no_exec = 1;
            return SECCOMP_RET_ALLOW;
        }
        return SECCOMP_RET_ERRNO | EPERM;
    }
    return SECCOMP_RET_ALLOW;
}
```

eBPF map to store states, i.e., whether execve has been called
Syscall count limiting

- Solution: Implement a state-based policy in eBPF

```c
unsigned int no_exec;
SEC("seccomp")
int exec_once(struct seccomp_data *ctx)
{
    if (ctx->nr == __NR_execve) {
        if (!no_exec) {
            no_exec = 1;
            return SECCOMP_RET_ALLOW;
        }
    }
    return SECCOMP_RET_ERRNO | EPERM;
}
return SECCOMP_RET_ALLOW;
```

If execve hasn't been called, allow and mark it as called
Syscall count limiting

• Solution: Implement a state-based policy in eBPF

```c
unsigned int no_exec;
SEC("seccomp")
int exec_once(struct seccomp_data *ctx)
{
    if (ctx->nr == __NR_execve) {
        if (!no_exec) {
            no_exec = 1;
            return SECCOMP_RET_ALLOW;
        }
        return SECCOMP_RET_ERRNO | EPERM;  
    }
    return SECCOMP_RET_ERRNO | EPERM;  
}
return SECCOMP_RET_ALLOW;
```
Syscall count limiting

- Solution: Implement a state-based policy in eBPF

```c
unsigned int no_exec;
SEC("seccomp")
int exec_once(struct seccomp_data *ctx)
{
    if (ctx->nr == __NR_execve) {
        if (!no_exec) {
            no_exec = 1;
            return SECCOMP_RET_ALLOW;
        }
        return SECCOMP_RET_ERRNO | EPERM;
    }
    return SECCOMP_RET_ALLOW;
}
```

Otherwise reject the syscall

Additional attack surface eliminated
Temporal syscall specialization [1]

• Restrict different syscalls at different phases of an application
Temporal syscall specialization [1]

- Restrict different syscalls at different phases of an application
  - E.g., webservers

Temporal syscall specialization [1]

- Restrict different syscalls at different phases of an application
  - E.g., web servers
Temporal syscall specialization [1]

- Restrict different syscalls at different phases of an application
  - E.g., webservers
- cBPF cannot enforce the policy precisely

---

Temporal syscall specialization [1]

- Restrict different syscalls at different phases of an application
  - E.g., webservers
- cBPF cannot enforce the policy **precisely**
- eBPF map can track application phases
  - Different policies for different phases

Syscall-flow-integrity protection [1]
Syscall-flow-integrity protection [1]

- Complements the concept of CFI with integrity for user-kernel transitions.
Syscall-flow-integrity protection [1]

• Complements the concept of CFI with integrity for user-kernel transitions.

• Extract allowed syscall transitions to construct a syscall state machine

Syscall-flow-integrity protection [1]

- Complements the concept of CFI with integrity for user-kernel transitions.
- Extract allowed syscall transitions to construct a syscall state machine
- Enforce such transitions at application runtime

Syscall-flow-integrity protection [1]

- Complements the concept of CFI with integrity for user-kernel transitions.
- Extract allowed syscall transitions to construct a syscall state machine.
- Enforce such transitions at application runtime.

Syscall-flow-integrity protection [1]

- Complements the concept of CFI with integrity for user-kernel transitions.
- Extract allowed syscall transitions to construct a syscall state machine
- Enforce such transitions at application runtime

Syscall-flow-integrity protection [1]

• Complements the concept of CFI with integrity for user-kernel transitions.

• Extract allowed syscall transitions to construct a syscall state machine

• Enforce such transitions at application runtime

Syscall-flow-integrity protection [1]

- Complements the concept of CFI with integrity for user-kernel transitions.
- Extract allowed syscall transitions to construct a syscall state machine
- Enforce such transitions at application runtime
- Construct syscall state machine as an eBPF map and use Seccomp for enforcement

How to make Seccomp-eBPF secure?

- Security model
- support for eBPF Seccomp filters under privileged eBPF
- IMA integration to enhance security
Security model of Seccomp-eBPF

- Principle: Reduce to security of Seccomp and eBPF subsystems
  - Seccomp-eBPF is **as secure as** the existing Seccomp and eBPF
Security model of Seccomp-eBPF

• Principle: Reduce to security of Seccomp and eBPF subsystems
  • Seccomp-eBPF is as secure as the existing Seccomp and eBPF

• Seccomp requires privileges for filter installation
  • requires either CAP_SYS_ADMIN or NO_NEW_PRIVS attribute on the process
Security model of Seccomp-eBPF

- Principle: Reduce to security of Seccomp and eBPF subsystems
  - Seccomp-eBPF is as secure as the existing Seccomp and eBPF

- Seccomp requires privileges for filter installation
  - requires either CAP_SYS_ADMIN or NO_NEW_PRIVS attribute on the process

- eBPF security is achieved by verification and privileges for program loading and helper access
Security model of Seccomp-eBPF

- Principle: Reduce to security of Seccomp and eBPF subsystems
  - Seccomp-eBPF is as secure as the existing Seccomp and eBPF

- Seccomp requires privileges for filter installation
  - requires either CAP_SYS_ADMIN or NO_NEW_PRIVS attribute on the process

- eBPF security is achieved by verification and privileges for program loading and helper access

- Seccomp-eBPF maintains the above security enforcement.
Reduce to eBPF security
Reduce to eBPF security

- Program loading
  - concerns context access and attach point
  - covered by Seccomp security
Reduce to eBPF security

• Program loading
  • concerns context access and attach point
  • covered by Seccomp security

• Same privilege requirement for helpers
  • Basic helpers are unprivileged
  • Tracing helpers still require CAP_BPF and CAP_PERFMON
Reduce to eBPF security

- Program loading
  - concerns context access and attach point
  - covered by Seccomp security
- Same privilege requirement for helpers
  - Basic helpers are unprivileged
  - Tracing helpers still require CAP_BPF and CAP_PERFMON
- eBPF does not have privilege requirement for maps
  - Add a new verifier hook to restrict map usage if desired
Unprivileged eBPF filters?

• Vulnerabilities in eBPF are uncovered in recent years
  • prompted community to discourage unprivileged eBPF
Unprivileged eBPF filters?

- Vulnerabilities in eBPF are uncovered in recent years
  - prompted community to discourage unprivileged eBPF
- Seccomp-eBPF is useful under privileged eBPF
Unprivileged eBPF filters?

• Vulnerabilities in eBPF are uncovered in recent years
  • prompted community to discourage unprivileged eBPF

• Seccomp-eBPF is useful under privileged eBPF
  • lots of virtualization applications run as root
  • privileged enough to utilize eBPF Seccomp filters
    • example: Docker
Harden Seccomp-eBPF filters

• Optionally disables unprivileged eBPF filters based on the BPF_UNPRIV_DEFAULT_OFF configuration
Harden Seccomp-eBPF filters

• Optionally disables unprivileged eBPF filters based on the BPF_UNPRIV_DEFAULT_OFF configuration

• Same privilege requirement as eBPF tracing program types
  • CAP_BPF + CAP_PERFMON when unprivileged filters are not allowed
  • eBPF filters are similar to tracing programs in many aspects
    • e.g., common set of helpers, changing function return value / process behavior
Potential security support from IMA

• If we can verify an eBPF Seccomp filter comes from a trusted source, it is always secure to load it into the kernel and use it
  • turns the security problem into a trust problem
Potential security support from IMA

- If we can verify an eBPF Seccomp filter comes from a trusted source, it is always secure to load it into the kernel and use it
  - turns the security problem into a trust problem

- The IMA subsystem checks signatures of files and verifies whether it comes from a trusted source
Potential security support from IMA

• If we can verify an eBPF Seccomp filter comes from a trusted source, it is always secure to load it into the kernel and use it
  • turns the security problem into a trust problem

• The IMA subsystem checks signatures of files and verifies whether it comes from a trusted source

• Idea: implementing IMA signature support for eBPF filters
  • The IMA community is welcome to the eBPF signature support
  • Refer to KP’s talk :)

Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022
Our eBPF Seccomp filter support
Our eBPF Seccomp filter support

• Based on Dhillon’s patch in 2018
Our eBPF Seccomp filter support

• Based on Dhillon’s patch in 2018
• A new program type: `BPF_PROG_TYPE_SECCOMP`
Our eBPF Seccomp filter support

• Based on Dhillon’s patch in 2018
• A new program type: BPF_PROG_TYPE_SECCOMP
• Same attach point as cBPF filters
Our eBPF Seccomp filter support

- Based on Dhillon’s patch in 2018
- A new program type: `BPF_PROG_TYPE_SECCOMP`
- Same attach point as cBPF filters
- Verifier hooks ensure correct access to the `seccomp_data` context and restrict the set of helpers for eBPF filters
Our eBPF Seccomp filter support

• Based on Dhillon’s patch in 2018
• A new program type: `BPF_PROG_TYPE_SECCOMP`
• Same attach point as cBPF filters
• Verifier hooks ensure correct access to the `seccomp_data` context and restrict the set of helpers for eBPF filters
• Security model: Reduce to Seccomp and eBPF
Container runtime integration

• Seccomp-eBPF can be easily integrated with container runtimes
• It is supported in crun,
  • A fast OCI-compliant runtime and the default for Podman
  • Credits to Giuseppe Scrivano (RedHat)
Container runtime integration

- Seccomp-eBPF can be easily integrated with container runtimes
- It is supported in crun,
  - A fast OCI-compliant runtime and the default for Podman
  - Credits to Giuseppe Scrivano (RedHat)

```bash
$ podman --runtime /usr/local/bin/crun run
   --annotation run.oci.seccomp_ebpf_file
      =ebpf_filter.o
```
Conclusion

• We revisit Seccomp-eBPF filters
  • Significantly improve programmability of system call security
  • Many real-world use cases
  • Security model can be well defined

• https://github.com/xlab-uiuc/seccomp-ebpf-upstream