Outline

12  Seccomp  12-1
12.1  Introduction and history  12-3
12.2  Seccomp filtering and BPF  12-10
12.3  The BPF virtual machine and BPF instructions  12-16
12.4  BPF filter return values  12-28
12.5  BPF programs  12-34
12.6  Checking the architecture  12-49
12.7  Discovering the system calls made by a program  12-55
12.8  Further details on seccomp filters  12-60
12.9  Caveats  12-69
12.10  Productivity aids (libseccomp and other tools)  12-73
12.11  Extended BPF (eBPF)  12-83
12.12  Applications and further information  12-87
12.13  Further details on BPF programs  12-90
12.14  Recent seccomp features  12-94
12.15  Audit logging of filter actions  12-98
### What is seccomp?

- Kernel provides large number of system calls
  - ≈400 system calls
- Each system call is a vector for attack against kernel
- Most programs use only small subset of available system
  calls
  - Remaining systems calls should never legitimately occur
  - If they do occur, perhaps it is because program has been compromised
- Seccomp = mechanism to restrict system calls that a
  process may make
  - Reduces attack surface of kernel
  - A key component for building application sandboxes
Introduction and history

- First version in Linux 2.6.12 (2005)
  - Filtering enabled via `/proc/PID/seccomp`
    - Writing “1” to file places process (irreversibly) in “strict” seccomp mode
  - Need CONFIG_SECCOMP
- **Strict mode**: only permitted system calls are `read()`, `write()`, `_exit()`, and `sigreturn()`
  - Note: `open()` not included (must open files before entering strict mode)
  - `sigreturn()` allows for signal handlers
- Other system calls ⇒ **SIGKILL**
- Designed to sandbox compute-bound programs that deal with untrusted byte code
  - Code perhaps exchanged via pre-created pipe or socket

---

Linux 2.6.23 (2007):

- `/proc/PID/seccomp` interface replaced by `prctl()` operations
- `prctl(PR_SET_SECCOMP, arg)` modifies caller’s seccomp mode
  - `SECCOMP_MODE STRICT`: limit syscalls as before
- `prctl(PR_GET_SECCOMP)` returns seccomp mode:
  - 0 ⇒ process is not in seccomp mode
  - Otherwise?
    - **SIGKILL (!)**
      - `prctl()` is not a permitted system call in “strict” mode
      - Who says kernel developers don’t have a sense of humor?
Introduction and history

- Linux 3.5 (July 2012) adds “filter” mode (AKA “seccomp2”)
  - `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, ...)
  - Can control which system calls are permitted to calling thread
    - Control based on system call number and argument values
  - Choice is controlled by user-defined filter—a BPF “program”
    - Berkeley Packet Filter (later)
  - Requires `CONFIG_SECCOMP_FILTER`
  - By now used in a range of tools
    - E.g., Chrome browser, OpenSSH, `vsftpd`, `systemd`, Firefox OS, Docker, LXC, Flatpak, Firejail, `strace`

Linux Security and Isolation APIs ©2020, Michael Kerrisk
Seccomp 12-7 §12.1

Introduction and history

- Linux 3.8 (2013):
  - The joke is getting old...
  - New `/proc/PID/status` Seccomp field exposes process seccomp mode (as a number)
    - 0  // SECCOMP_MODE_DISABLED
    - 1  // SECCOMP_MODE_STRICT
    - 2  // SECCOMP_MODE_FILTER
  - Process can, without fear, read from this file to discover its own seccomp mode
    - But, must have previously obtained a file descriptor...
Introduction and history

- **Linux 3.17 (2014):**
  - `seccomp()` system call added
  - (Rather than further multiplexing of `prctl()`)  
  - `seccomp(2)` provides superset of `prctl(2)` functionality
  - Can synchronize all threads to same filter tree
  - Useful, e.g., if some threads created by start-up code before application has a chance to install filter(s)

- **Linux 4.14 (2017):**
  - Audit logging of seccomp actions
  - Interfaces to discover what seccomp features are supported by kernel
  - Wider range of “actions” can be returned by BPF filters

- **Linux 5.0 (March 2019):**
  - New action: notification to user-space process
Outline

12 Seccomp 12-1
12.1 Introduction and history 12-3
12.2 Seccomp filtering and BPF 12-10
12.3 The BPF virtual machine and BPF instructions 12-16
12.4 BPF filter return values 12-28
12.5 BPF programs 12-34
12.6 Checking the architecture 12-49
12.7 Discovering the system calls made by a program 12-55
12.8 Further details on seccomp filters 12-60
12.9 Caveats 12-69
12.10 Productivity aids (libseccomp and other tools) 12-73
12.11 Extended BPF (eBPF) 12-83
12.12 Applications and further information 12-87
12.13 Further details on BPF programs 12-90
12.14 Recent seccomp features 12-94
12.15 Audit logging of filter actions 12-98

Seccomp filtering overview

- Allows filtering based on system call number and argument (register) values
  - Pointers are **not** dereferenced
- Steps:
  1. Construct filter program that specifies permitted system calls
  - Filters expressed as BPF (Berkeley Packet Filter) programs
  2. Install filter using `seccomp()` or `prctl()`
  3. `exec()` new program or invoke function inside dynamically loaded shared library (plug-in)
- Once installed, **every syscall triggers execution of filter**
  - Installed filters **can’t** be removed
  - Filter == declaration that we don’t trust subsequently executed code
BPF origins

- Seccomp filters are expressed using BPF (Berkeley Packet Filter) syntax
- BPF originally devised (in 1992) for *tcpdump*
  - Monitoring tool to display packets passing over network
- Volume of network traffic is enormous ⇒ must filter for packets of interest
- BPF allows **in-kernel selection of packets**
  - Filtering based on fields in packet header
  - Filtering in kernel more efficient than filtering in user space
  - Unwanted packets are **discarded early**
  - ⇒ Avoids passing every packet over kernel-user-space boundary

BPF virtual machine

- BPF defines a **virtual machine** (VM) that can be implemented inside kernel
- VM characteristics:
  - **Simple instruction set**
    - Small set of instructions
    - All instructions are same size (64 bits)
    - Implementation is simple and fast
  - Only **branch-forward** instructions
    - Programs are directed acyclic graphs (DAGs)
  - Easy to verify validity/safety of programs
    - Program completion is guaranteed (DAGs)
    - Simple instruction set ⇒ can verify opcodes and arguments
    - Can detect dead code
    - Can verify that program completes via a “return” instruction
    - BPF filter programs are limited to 4096 instructions
Generalizing BPF

- BPF originally designed to work with network packet headers
- Seccomp2 developers realized BPF could be generalized to solve different problem: filtering of system calls
  - Same basic task: test-and-branch processing based on content of a small set of memory locations
Key features of BPF virtual machine

- Accumulator register (32-bit)
- Data area (data to be operated on)
  - In seccomp context: data area describes system call
- All instructions are 64 bits, with a fixed format
  - Expressed as a C structure, that format is:

```c
struct sock_filter {
    __u16 code;    /* Filter code (opcode)*/
    __u8  jt;      /* Jump true */
    __u8  jf;      /* Jump false */
    __u32 k;       /* Multiuse field (operand) */
};
```

- See `<linux/filter.h>` and `<linux/bpf_common.h>`
- No state is preserved between BPF program invocations
  - E.g., can’t intercept n’th syscall of a particular type
BPF instruction set

Instruction set includes:

- Load instructions (**BPF_LD**)
- Store instructions (**BPF_ST**)
  - There is a “working memory” area where info can be stored (not persistent)
- Jump instructions (**BPF_JMP**)
- Arithmetic/logic instructions (**BPF_ALU**)
  - **BPF_ADD**, **BPF_SUB**, **BPF_MUL**, **BPF_DIV**, **BPF_MOD**, **BPF_NEG**
  - **BPF_OR**, **BPF_AND**, **BPF_XOR**, **BPF_LSH**, **BPF_RSH**
- Return instructions (**BPF_RET**)
  - Terminate filter processing
  - Report a status telling kernel what to do with syscall

BPF jump instructions

- Conditional and unconditional jump instructions provided
- Conditional jump instructions consist of
  - **Opcode** specifying condition to be tested
  - **Value** to test against
  - Two jump targets
    - **jt**: target if condition is true
    - **jf**: target if condition is false
- Conditional jump instructions:
  - **BPF_JEQ**: jump if equal
  - **BPF_JGT**: jump if greater
  - **BPF_JGE**: jump if greater or equal
  - **BPF_JSET**: bit-wise AND + jump if nonzero result
  - **jf** target ⇒ no need for **BPF_{JNE, JLT, JLE, JCLEAR}**
BPF jump instructions

- Targets are expressed as relative offsets in instruction list
  - $0 = \text{no jump (execute next instruction)}$
  - $jt$ and $jf$ are 8 bits $\Rightarrow$ 255 maximum offset for conditional jumps
- Unconditional BPF_JA (“jump always”) uses $k$ as offset, allowing much larger jumps

Seccomp BPF data area

- Seccomp provides data describing syscall to filter program
  - Buffer is read-only
    - I.e., seccomp filter can’t change syscall or syscall arguments
  - Can be expressed as a C structure...
Seccomp BPF data area

```
struct seccomp_data {
    int  nr;            /* System call number */
    __u32 arch;         /* AUDIT_ARCH_* value */
    __u64 instruction_pointer; /* CPU IP */
    __u64 args[6];     /* System call arguments */
};
```

- **nr**: system call number (architecture-dependent); 4-byte `int`
- **arch**: identifies architecture
  - Constants defined in `<linux/audit.h>`
    - `AUDIT_ARCH_X86_64`, `AUDIT_ARCH_ARM`, etc.
- **instruction_pointer**: CPU instruction pointer
- **args**: system call arguments
  - System calls have maximum of six arguments
  - Number of elements used depends on system call

Building BPF instructions

- Obviously, one could code BPF instructions numerically by hand
- But, header files define symbolic constants and convenience macros (`BPF_STMT()`, `BPF_JUMP()`) to ease the task

```c
#define BPF_STMT(code, k)  
    { (unsigned short)(code), 0, 0, k } 
#define BPF_JUMP(code, k, jt, jf)  
    { (unsigned short)(code), jt, jf, k } 
```

- These macros just plug values together to form structure initializer
Building BPF instructions: examples

- **Load architecture number into accumulator**

```c
BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, arch)))
```

- **Opcode here is constructed by ORing three values together:**
  - **BPF_LD**: load
  - **BPF_W**: operand size is a word (4 bytes)
  - **BPF_ABS**: address mode specifying that source of load is data area (containing system call data)
  - See `<linux/bpf_common.h>` for definitions of opcode constants

- **Operand is architecture field of data area**
  - `offsetof()` yields byte offset of a field in a structure

---

Building BPF instructions: examples

- **Test value in accumulator**

```c
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 1, 0)
```

- **BPF_JMP | BPF_JEQ**: jump with test on equality
- **BPF_K**: value to test against is in generic multiuse field \(k\)
- **\(k\) contains value AUDIT_ARCH_X86_64**
- **\(jt\) value is 1, meaning skip one instruction if test is true**
- **\(jf\) value is 0, meaning skip zero instructions if test is false**
  - i.e., continue execution at following instruction
Building BPF instructions: examples

- Return value that causes kernel to kill process
  
  \[
  \text{BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)}
  \]

- Arithmetic/logic instruction: add one to accumulator
  
  \[
  \text{BPF_STMT (BPF_ALU | BPF_ADD | BPF_K, 1)}
  \]

- Arithmetic/logic instruction: right shift accumulator 12 bits
  
  \[
  \text{BPF_STMT (BPF_ALU | BPF_RSH | BPF_K, 12)}
  \]
Filter return value

- Once a filter is installed, each system call is tested against filter
- Seccomp filter must return a value to kernel indicating whether system call is permitted
  - Otherwise EINVAL when attempting to install filter
- Return value is 32 bits, in two parts:
  - Most significant 16 bits (SECCOMP_RET_ACTION_FULL mask) specify an action to kernel
  - Least significant 16 bits (SECCOMP_RET_DATA mask) specify “data” for return value

```c
#define SECCOMP_RET_ACTION_FULL 0xffff0000U
#define SECCOMP_RET_DATA 0x0000ffffU
```
Filter return action (1)

Filter return action component is one of:

- **SECCOMP_RET_ALLOW**: system call is allowed to execute
- **SECCOMP_RET_KILL_PROCESS** (since Linux 4.14): process (all threads) is immediately killed
  - Terminated *as though* process had been killed with **SIGSYS**
  - There is no actual **SIGSYS** signal delivered, but...
  - To parent (via `wait()`) it appears child was killed by **SIGSYS**
  - Core dump is also produced
- **SECCOMP_RET_KILL_THREAD** (alias added in Linux 4.14): thread (i.e., task, not process) is immediately killed
  - Terminated *as though* thread had been killed with **SIGSYS**
  - If only thread in process, core dump is also produced
  - **SECCOMP_RET_KILL_THREAD**

Filter return action (2)

- **SECCOMP_RET_ERRNO**: return an error from system call
  - System call is not executed
  - Value in **SECCOMP_RET_DATA** is returned in **errno**
- **SECCOMP_RET_USER_NOTIF** (since Linux 5.0): send notification to user-space “tracing” process
  - System call is **not** executed
  - Notified process (the “tracer”):
    - Receives syscall info (same as BPF filter) + PID of filtered process (the “target”)
    - Can use received info to (for example) inspect arguments of “target” syscall (via `/proc/PID/mem`)
    - Can take appropriate action (e.g., perform operation on behalf of “target”)
    - Provides (fake) success/error return value for syscall
  - See `seccomp(2) + seccomp/seccomp_user_notification.c`
  - Added for some container use cases, but other uses are possible
Filter return action (3)

- **SECCOMP_RET_TRACE**: attempt to notify `ptrace()` tracer before making syscall
  - Gives tracing process a chance to assume control
    - If there is no tracer, syscall fails with `ENOSYS` error
  - `strace(1)` uses this to speed tracing (since 2018)
  - See `seccomp(2)`

- **SECCOMP_RET_TRAP**: calling thread is sent `SIGSYS` signal
  - Can catch this signal; see `seccomp(2)` for more details
  - Example: `seccomp/seccomp_trap_sigsys.c`

- **SECCOMP_RET_LOG** (since Linux 4.14): allow + log syscall
  - System call is allowed, and also logged to audit log
    - `/var/log/audit/audit.log`; `ausearch(8)`
  - Useful during filter development (later...)
### Installing a BPF program

- A process installs a filter for itself using one of:
  - `seccomp(SECCOMP_SET_MODE_FILTER, flags, &fprog)`
    - Only since Linux 3.17
  - `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &fprog)`
- `&fprog` is a pointer to a BPF program:

```c
struct sock_fprog {
    unsigned short len;  /* Number of instructions */
    struct sock_filter *filter;
    /* Pointer to program (array of instructions) */
};
```
Installing a BPF program

To install a filter, one of the following must be true:

- Caller is privileged (has CAP_SYS_ADMIN in its user namespace)
- Caller has to set the no_new_privs attribute:

```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
```

- Causes set-UID/set-GID bit / file capabilities to be ignored on subsequent execve() calls
  - Once set, no_new_privs can’t be unset
  - Per-thread attribute
- Prevents possibility of attacker starting privileged program and manipulating it to misbehave using a seccomp filter
- ! no_new_privs && ! CAP_SYS_ADMIN ⇒ seccomp() / prctl(PR_SET_SECCOMP) fails with EACCES

Example: seccomp/seccomp_deny_open.c

```c
int main(int argc, char *argv[]) {
    prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
    install_filter();
    open("/tmp/a", O_RDONLY);
    printf("We shouldn't see this message\n");
    exit(EXIT_SUCCESS);
}
```

Program installs a filter that prevents open() and openat() being called, and then calls open():

- Set no_new_privs bit
- Install seccomp filter
- Call open()
static void install_filter(void) {
    struct sock_filter filter[] = {
        /* Architecture-check code not shown */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr))), ...
    };

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 1, 0),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
};

- BPF filter program consists of a series of sock_filter structs
- For now we ignore some BPF code that checks the architecture that BPF program is executing on
  - ⚠️ This is an essential part of every BPF filter program
- Load system call number into accumulator
- (BPF program continues on next slide)

- Test if system call number matches __NR_open
  - True: advance two instructions ⇒ kill process
  - False: advance 0 instructions ⇒ next test
- Test if system call number matches __NR_openat
  - True: advance one instruction ⇒ kill process
  - False: advance 0 instructions ⇒ allow syscall
- (Note: creat() + open_by_handle_at() are still not filtered)
Example: seccomp/seccomp_deny_open.c

```c
struct sock_fprog prog = {
    .len = (unsigned short) (sizeof(filter) /
                        sizeof(filter[0])),
    .filter = filter,
};
seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog);
```

- Construct argument for `seccomp()`
- Install filter

Upon running the program, we see:

```
$ ./seccomp_deny_open
Bad system call  # Message printed by shell
$ echo $?        # Display exit status of last command
159
```

- "Bad system call" printed by shell, because it looks like its child was killed by `SIGSYS`
- Exit status of 159 (== 128 + 31) also indicates termination as though killed by `SIGSYS`
  - Exit status of process killed by signal is 128 + `signum`
  - `SIGSYS` is signal number 31 on this architecture
A more sophisticated example

Filter based on *flags* argument of `open()` / `openat()`

- 0_CREAT specified ⇒ kill process
- 0_WRONLY or 0_RDWR specified ⇒ cause call to fail with ENOTSUP error

*flags* is arg. 2 of `open()`, and arg. 3 of `openat()`:

```c
int open(const char *pathname, int flags, ...);
int openat(int dirfd, const char *pathname, int flags, ...);
```

*flags* serves exactly the same purpose for both calls

---

```
struct sock_filter filter[] = {
    /* Architecture-check code not shown */
    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, 
        (offsetof(struct seccomp_data, nr))),
    ...
```

Load system call number
Example: seccomp/seccomp_control_open.c

BPF_JUMP (BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
BPF_JUMP (BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 3, 0),
BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

/* Load open() flags */
BPF_STMT (BPF_LD | BPF_W | BPF_ABS,
          (offsetof(struct seccomp_data, args[1]))),
BPF_JUMP (BPF_JMP | BPF_JA, 1, 0, 0),

/* Load openat() flags */
BPF_STMT (BPF_LD | BPF_W | BPF_ABS,
          (offsetof(struct seccomp_data, args[2]))),

- Allow system calls other than open() / openat()
- For open(), load flags argument (args[1]) into accumulator, and then jump over next instruction
- For openat(), load flags argument (args[2]) into accumulator

Example: seccomp/seccomp_control_open.c

BPF_JUMP (BPF_JMP | BPF_JSET | BPF_K, O_CREAT, 0, 1),
BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
BPF_JUMP (BPF_JMP | BPF_JSET | BPF_K, O_WRONLY | O_RDWR, 0, 1),
BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_ERRNO | ENOTSUP),
BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_ALLOW)

- Test if O_CREAT bit is set in flags
  - True: skip 0 instructions ⇒ kill process
  - False: skip 1 instruction
- Test if O_WRONLY or O_RDWR is set in flags
  - True: cause call to fail with ENOTSUP error in errno
  - False: allow call to proceed
```c
int main(int argc, char **argv) {
    prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
    install_filter();

    if (open("/tmp/a", O_RDONLY) == -1)
        perror("open1");
    if (open("/tmp/a", O_WRONLY) == -1)
        perror("open2");
    if (open("/tmp/a", O_RDWR) == -1)
        perror("open3");
    if (open("/tmp/a", O_CREAT | O_RDWR, 0600) == -1)
        perror("open4");

    exit(EXIT_SUCCESS);
}
```

- Test `open()` calls with various flags

```
Test open() calls with various flags
```

```
$ ./seccomp_control_open
open2: Operation not supported
open3: Operation not supported
Bad system call
$ echo $?
159
```

- First `open()` succeeded
- Second and third `open()` calls failed
  - Kernel produced ENOTSUP error for call
- Fourth `open()` call caused process to be killed

```
First open() succeeded
Second and third open() calls failed
  - Kernel produced ENOTSUP error for call
Fourth open() call caused process to be killed
```