Using seccomp to limit the kernel attack surface

Michael Kerrisk, man7.org © 2018
mtk@man7.org

Embedded Linux Conference Europe 2018
22 October 2018, Edinburgh, Scotland
Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
    - Maintainer email: mtk.manpages@gmail.com
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
- Author of a book on the Linux programming interface
  - http://man7.org/tlpi/
- Trainer/writer/engineer
  - Lots of courses at http://man7.org/training/
- Email: mtk@man7.org
  - Twitter: @mkerrisk
## Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Seccomp</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Seccomp filtering and BPF</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Checking the architecture</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>BPF filter return values</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>BPF programs</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>Another example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Further details on seccomp filters</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Caveats</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Productivity aids (libseccomp and other tools)</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td>Applications and further information</td>
<td>54</td>
</tr>
</tbody>
</table>
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 occur
  - If they do occur, perhaps it is because program has been compromised
- Seccomp = mechanism to restrict the system calls that a process may make
  - Reduces attack surface of kernel
  - A key component for building application sandboxes
Development 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

- **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 \(\Rightarrow\) SIGKILL

- Designed to sandbox compute-bound programs that deal with untrusted byte code
  - Code perhaps exchanged via pre-created pipe or socket
Development history

- Linux 3.5 (2012) adds “filter” mode (AKA “seccomp2”)
  - `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, ...)`
  - Can control which system calls are permitted to caller
    - Control based on system call number and argument values
  - By now used in a range of tools
    - E.g., Chrome browser, OpenSSH, `vsftpd`, `systemd`, Firefox OS, Docker, LXC, Flatpak, Firejail

- Linux 3.17 (2014):
  - `seccomp()` system call added
    - (Rather than further multiplexing of `prctl()`)
  - `seccomp()` provides superset of `prctl(2)` functionality

- And work is ongoing...
  - E.g., several features added in Linux 4.14
Outline

1 Introduction 3
2 Introduction to Seccomp 5
3 Seccomp filtering and BPF 9
4 The BPF virtual machine and BPF instructions 13
5 Checking the architecture 23
6 BPF filter return values 25
7 BPF programs 28
8 Another example 36
9 Further details on seccomp filters 43
10 Caveats 47
11 Productivity aids (libseccomp and other tools) 49
12 Applications and further information 54
Seccomp filtering overview

- Fundamental idea: filter system calls based on syscall number and argument (register) values
  - Pointers are not dereferenced
- To employ seccomp, the user-space program does following:
  1. **Construct filter program** that specifies permitted syscalls
     - Filters expressed as BPF (Berkeley Packet Filter) programs
  2. **Install filter program into kernel** using `seccomp()`/`prctl()`
  3. **Execute untrusted code**: `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 as BPF (Berkeley Packet Filter) programs
- BPF originally devised (in 1992) for *tcpdump*
  - Monitoring tool to display packets passing over network
- Volume of network traffic is enormous $\Rightarrow$ 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 packet are *discarded early*
  - Avoid passing *every* packet over kernel-user-space boundary
- Seccomp $\Rightarrow$ generalize BPF model to filter on syscall info
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 BPF programs
    - Program completion is guaranteed (DAGs)
    - Simple instruction set \( \Rightarrow \) 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Introduction to Seccomp</td>
<td>5</td>
</tr>
<tr>
<td>3. Seccomp filtering and BPF</td>
<td>9</td>
</tr>
<tr>
<td><strong>4. The BPF virtual machine and BPF instructions</strong></td>
<td><strong>13</strong></td>
</tr>
<tr>
<td>5. Checking the architecture</td>
<td>23</td>
</tr>
<tr>
<td>6. BPF filter return values</td>
<td>25</td>
</tr>
<tr>
<td>7. BPF programs</td>
<td>28</td>
</tr>
<tr>
<td>8. Another example</td>
<td>36</td>
</tr>
<tr>
<td>9. Further details on seccomp filters</td>
<td>43</td>
</tr>
<tr>
<td>10. Caveats</td>
<td>47</td>
</tr>
<tr>
<td>11. Productivity aids (libseccomp and other tools)</td>
<td>49</td>
</tr>
<tr>
<td>12. Applications and further information</td>
<td>54</td>
</tr>
</tbody>
</table>
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;   /* Generic multiuse field (operand) */
};
```

- See `<linux/filter.h>` and `<linux/bpf_common.h>`
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
  - Working memory is not persistent between filter invocations
- 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 == no jump (execute next instruction)
  - jt and jf are 8 bits \(\Rightarrow\) 255 maximum offset for conditional jumps
- Unconditional BPF_JA ("jump always") uses \(k\) (operand) 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)
- **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

- Return value that causes kernel to kill process

  ```c
  BPF_STMT(BPF_RET | BPF_K,
           SECCOMP_RET_KILL_PROCESS)
  ```
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Seccomp</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Seccomp filtering and BPF</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Checking the architecture</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>BPF filter return values</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>BPF programs</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>Another example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Further details on seccomp filters</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Caveats</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Productivity aids (libseccomp and other tools)</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td>Applications and further information</td>
<td>54</td>
</tr>
</tbody>
</table>
Checking the architecture

- Checking architecture value should be first step in any BPF program
- Syscall numbers differ across architectures!
  - May have built seccomp BPF BLOB for one architecture, but accidentally load it on different architecture
- Hardware may support multiple system call conventions
  - E.g. modern x86 hardware supports three(!) architecture+ABI conventions
    - *During life of process syscall ABI may change* (as new binaries are execed)
    - But, *scope of BPF filter is lifetime of process*
- System call numbers may differ under each convention
- For an example, see seccomp/seccomp_multiarch.c
Outline

1 Introduction 3
2 Introduction to Seccomp 5
3 Seccomp filtering and BPF 9
4 The BPF virtual machine and BPF instructions 13
5 Checking the architecture 23
6 BPF filter return values 25
7 BPF programs 28
8 Another example 36
9 Further details on seccomp filters 43
10 Caveats 47
11 Productivity aids (libseccomp and other tools) 49
12 Applications and further information 54
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 0xffffffffU
#define SECCOMP_RET_DATA 0x0000ffffU
```
Filter return action

Various possible filter return actions, including:

- **SECCOMP_RET_ALLOW**: system call is allowed to execute
- **SECCOMP_RET_KILL_PROCESS**: process (all threads) is 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
- **SECCOMP_RET_KILL_THREAD**: calling thread is killed
  - Terminated *as though* thread had been killed with SIGSYS
- **SECCOMP_RET_ERRNO**: return an error from system call
  - System call is not executed
  - Value in `SECCOMP_RET_DATA` is returned in `errno`
- Also: **SECCOMP_RET_TRACE**, **SECCOMP_RET_TRAP**, **SECCOMP_RET_LOG**
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
- 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()`
Example: seccomp/seccomp_deny_open.c

```c
static void install_filter(void) {
    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,
          AUDIT_ARCH_X86_64, 1, 0),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
    ...}
```

- Initialize array (of 64-bit structs) containing filter program
- Load architecture into accumulator
- Test if architecture value matches AUDIT_ARCH_X86_64
  - True: jump forward one instruction (i.e., skip next instr.)
  - False: skip no instructions
- Kill process on architecture mismatch
- (BPF program continues on next slide)
Example: seccomp/seccomp_deny_open.c

```
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)
```

- Load system call number into accumulator
- 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
Example: seccomp/seccompdenyopen.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_den[y]_open
Bad system call   # Message printed by shell
$ echo $?        # Display exit status of last command
159
```

- “Bad system call” indicates process 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
Example: seccomp/seccomp_control_open.c

```c
definition

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,
              AUDIT_ARCH_X86_64, 1, 0),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
    BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
             (offsetof(struct seccomp_data, nr))),
}

- Load architecture and test for expected value
- Load system call number
```
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]))),

- (Syscall number is already in accumulator)
- 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

```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 & SECCOMP_RET_DATA)),

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
Example: seccomp/seccomp_control_open.c

```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
Example: `seccomp/seccomp_control_open.c`

```
$ ./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
Outline

1 Introduction 3
2 Introduction to Seccomp 5
3 Seccomp filtering and BPF 9
4 The BPF virtual machine and BPF instructions 13
5 Checking the architecture 23
6 BPF filter return values 25
7 BPF programs 28
8 Another example 36
9 Further details on seccomp filters 43
10 Caveats 47
11 Productivity aids (libseccomp and other tools) 49
12 Applications and further information 54
Installing multiple filters

- If existing filters permit `prctl()` or `seccomp()`, further filters can be installed
  - 32k maximum for total instructions in all filters
- **All** filters are always executed, in reverse order of registration
- Each filter yields a return value
- Value returned to kernel is first seen action of highest priority (along with accompanying data)
  - `SECCOMP_RET_KILL_PROCESS` (highest priority)
  - `SECCOMP_RET_KILL_THREAD` (SECCOMP_RET_KILL)
  - `SECCOMP_RET_TRAP`
  - `SECCOMP_RET_ERRNO`
  - `SECCOMP_RET_TRACE`
  - `SECCOMP_RET_LOG`
  - `SECCOMP_RET_ALLOW` (lowest priority)
fork() and execve() semantics

- If seccomp filters permit fork() or clone(), then child inherits parent’s filters
- If seccomp filters permit execve(), then filters are preserved across execve()
Cost of filtering, construction of filters

- Installed BPF filter(s) are executed for every system call
  - ⇒ there’s a performance cost

- Example on x86-64:
  - Use our “deny open” seccomp filter
    - Requires 6 BPF instructions / permitted syscall
  - Call `getppid()` repeatedly (one of cheapest syscalls)
  - +25% execution time (with JIT compiler disabled)
    - (Looks relatively high because `getppid()` is a cheap syscall)
    - (And it’s +25% on top of timings on kernel without Spectre/Meltdown mitigations enabled)

- Obviously, order of filtering rules can affect performance
  - Construct filters so that most common cases yield shortest execution paths
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2  Introduction to Seccomp</td>
<td>5</td>
</tr>
<tr>
<td>3  Seccomp filtering and BPF</td>
<td>9</td>
</tr>
<tr>
<td>4  The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5  Checking the architecture</td>
<td>23</td>
</tr>
<tr>
<td>6  BPF filter return values</td>
<td>25</td>
</tr>
<tr>
<td>7  BPF programs</td>
<td>28</td>
</tr>
<tr>
<td>8  Another example</td>
<td>36</td>
</tr>
<tr>
<td>9  Further details on seccomp filters</td>
<td>43</td>
</tr>
<tr>
<td>10 Caveats</td>
<td>47</td>
</tr>
<tr>
<td>11 Productivity aids (libseccomp and other tools)</td>
<td>49</td>
</tr>
<tr>
<td>12 Applications and further information</td>
<td>54</td>
</tr>
</tbody>
</table>
Caveats

- Adding a seccomp filter can **cause** bugs in application:
  - What if filter disallows a syscall that should have been allowed?
    - ⇒ **causes a legitimate application action to fail**
  - These buggy filters may be hard to find in testing, especially in rarely exercised code paths

- Filtering is based on **syscall numbers**, but **applications normally call C library wrappers** (not direct syscalls)
  - Wrapper function behavior may change across glibc versions or vary across architectures
    - E.g., in glibc 2.26, the `open()` wrapper switched from using `open(2)` to using `openat(2)` (and don’t forget `creat(2)`)
  - See https://lwn.net/Articles/738694/, *The inherent fragility of Seccomp*
Tools: libseccomp

- High-level API for kernel creating seccomp filters
  - https://github.com/seccomp/libseccomp
  - Initial release: 2012
- Simplifies various aspects of building filters
  - Eliminates tedious/error-prone tasks such as changing branch instruction counts when instructions are inserted
  - Abstract architecture-dependent details out of filter creation
  - Don’t have full control of generated code, but can give hints about which system calls to prioritize in generated code
    - seccomp_syscall_priority()
  - http://lwn.net/Articles/494252/
- Fully documented with man pages that contain examples (!)
libseccomp example (seccomp/libseccomp_demo.c)

```c
scmp_filter_ctx ctx;
ctx = seccomp_init(SCMP_ACT_ALLOW);
seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EPERM),
                 SCMP_SYS(clone), 0);
seccomp_rule_add(ctx, SCMP_ACT_ERRNO(ENOTSUP),
                 SCMP_SYS(fork), 0);
seccomp_load(ctx);

if (fork() != -1)
    fprintf(stderr, "fork() succeeded?!\n");
else
    perror("fork");
```

- Create seccomp filter state whose default action is to allow every syscall
- Disallow `clone()` and `fork()`, with different errors
- Load filter into kernel
- Try calling `fork()`
Example run (seccomp/libseccomp_demo.c)

```
$ ./libseccomp_demo
fork: Operation not permitted
```

- `fork()` fails, as expected
- EPERM error ⇒ `fork()` wrapper in glibc calls `clone()` (!)
Other tools

- **bpf** (BPF compiler)
  - Compiles assembler-like BPF programs to byte code
  - Part of *netsniff-ng* project ([http://netsniff-ng.org/](http://netsniff-ng.org/))

- In-kernel JIT (just-in-time) compiler
  - Compiles BPF binary to native machine code at load time
    - Execution speed up of 2x to 3x (or better, in some cases)
  - Disabled by default; enable by writing “1” to `/proc/sys/net/core/bpf_jit_enable`
    - Some distros build kernels with `CONFIG_BPF_JIT_ALWAYS_ON` option (available since Linux 4.15), which makes `bpf_jit_enable` immutably 1
  - See `bpf(2)` man page
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2  Introduction to Seccomp</td>
<td>5</td>
</tr>
<tr>
<td>3  Seccomp filtering and BPF</td>
<td>9</td>
</tr>
<tr>
<td>4  The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5  Checking the architecture</td>
<td>23</td>
</tr>
<tr>
<td>6  BPF filter return values</td>
<td>25</td>
</tr>
<tr>
<td>7  BPF programs</td>
<td>28</td>
</tr>
<tr>
<td>8  Another example</td>
<td>36</td>
</tr>
<tr>
<td>9  Further details on seccomp filters</td>
<td>43</td>
</tr>
<tr>
<td>10 Caveats</td>
<td>47</td>
</tr>
<tr>
<td>11 Productivity aids (libseccomp and other tools)</td>
<td>49</td>
</tr>
<tr>
<td>12 Applications and further information</td>
<td>54</td>
</tr>
</tbody>
</table>
Applications

Possible applications:

- Building sandboxed environments
  - Whitelisting usually safer than blacklisting
    - Default treatment: block all system calls
    - Then allow only a limited set of syscall / argument combinations
  - Various examples mentioned earlier
    - E.g., default Docker profile restricts various syscalls;
      chromium browser sandboxes rendering processes, which deal with untrusted inputs

- Failure-mode testing
  - Place application in environment where unusual / unexpected failures occur
  - Blacklist certain syscalls / argument combinations to generate failures
Resources

- Kernel source files:
  - Documentation/userspace-api/seccomp_filter.rst
  - Documentation/networking/filter.txt BPF VM in detail
- http://outflux.net/teach-seccomp/
- seccomp(2) man page
- “Seccomp sandboxes and memcached example”
  - blog.viraptor.info/post/seccomp-sandboxes-and-memcached-example-part-1
  - blog.viraptor.info/post/seccomp-sandboxes-and-memcached-example-part-2
- https://lwn.net/Articles/656307/
  - Write-up of a version of this presentation...
Thanks!

Michael Kerrisk  mtk@man7.org  @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/