Using seccomp to limit the kernel attack surface

© 2016 Michael Kerrisk
man7.org Training and Consulting
http://man7.org/training/
@mkerrisk mtk@man7.org

5 October 2016
Berlin, Germany
Outline

1 Introduction and history
2 Seccomp filtering and BPF
3 Constructing seccomp filters
4 BPF programs
5 Further details on seccomp filters
6 Applications, tools, and further information
Who am I?

- Maintainer of Linux man-pages (since 2004)
  - Documents kernel-user-space + C library APIs
    - ~1000 manual pages
- API review, testing, and documentation
  - API design and design review
  - Lots of testing, lots of bug reports, a few kernel patches
- “Day job”: programmer, trainer, writer
1 Introduction and history
2 Seccomp filtering and BPF
3 Constructing seccomp filters
4 BPF programs
5 Further details on seccomp filters
6 Applications, tools, and further information
What is seccomp?

- Kernel provides large number of systems calls
  - \( \approx 400 \) system calls
- Each system call is a vector for attack against kernel
- Most programs use only small subset of available system calls
- Seccomp = mechanism to restrict system calls that a process may make
  - Reduces attack surface of kernel
  - A key component for building application sandboxes
Outline

- History of seccomp
- Basics of seccomp operation
- Creating and installing BPF filters (AKA “seccomp2”)
  - Mostly: look at hand-coded BPF filter programs, to gain fundamental understanding of how seccomp works
  - Briefly note some productivity aids for coding BPF programs
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
Introduction and history

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?
Linux 3.5 (2012) adds “filter” mode (AKA “seccomp2”)

- prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, ...)
- Can control which system calls are permitted,
  - 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
Linux 3.8 (2013):
- The joke is getting old...
- New /proc/PID/status Seccomp field exposes process seccomp mode (as a number)

<table>
<thead>
<tr>
<th>Number</th>
<th>Seccomp Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SECCOMP_MODE_DISABLED</td>
</tr>
<tr>
<td>1</td>
<td>SECCOMP_MODE_STRICT</td>
</tr>
<tr>
<td>2</td>
<td>SECCOMP_MODE_FILTER</td>
</tr>
</tbody>
</table>

- 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()`)
- 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)
Outline

1 Introduction and history
2 Seccomp filtering and BPF
3 Constructing seccomp filters
4 BPF programs
5 Further details on seccomp filters
6 Applications, tools, and further information
Seccomp filtering and BPF

- Seccomp filtering available since Linux 3.5
- Allows filtering based on system call number and argument (register) values
  - Pointers are **not** dereferenced
- Filters expressed using BPF (Berkeley Packet Filter) syntax
- Filters installed using `seccomp()` or `prctl()`
  1. Construct and install BPF filter
  2. `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

- 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 packet 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
    - 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 $\Rightarrow$ 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

Seccomp: limiting the kernel attack surface  ContainerCon.eu 2016  Seccomp filtering and BPF
Generalizing BPF

- BPF originally designed to work with network packet headers
- Seccomp 2 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
- Further generalization ("extended BPF"; see `ebpf(2)`) is ongoing
  - Linux 3.18: adding filters to kernel tracepoints
  - Linux 3.19: adding filters to raw sockets
  - Linux 4.4: filtering of `perf` events
  - Linux 4.5: use cBPF or eBPF program to distribute packets to `SO_REUSEPORT` group of sockets
Outline

1 Introduction and history
2 Seccomp filtering and BPF
3 Constructing seccomp filters
4 BPF programs
5 Further details on seccomp filters
6 Applications, tools, and further information
Basic features of BPF virtual machine

- Accumulator register
- Data area (data to be operated on)
  - In seccomp context: data area describes system call
- Implicit program counter
  - (Recall: all instructions are same size)
- Instructions contained in structure of this form:

```c
struct sock_filter {
    /* Filter block */
    __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
- Store instructions
- Jump instructions
- Arithmetic/logic instructions
  - ADD, SUB, MUL, DIV, MOD, NEG
  - OR, AND, XOR, LSH, RSH
- Return instructions
  - 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:
  - **JEQ**: jump if equal
  - **JGT**: jump if greater
  - **JGE**: jump if greater or equal
  - **JSET**: bit-wise AND + jump if nonzero result
  - `jf` target ⇒ no need for **JNE**, **JLT**, **JLE**, and **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 ⇒ 255 maximum offset for conditional jumps
- Unconditional 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**
- Format (expressed as C struct):

```c
struct seccomp_data {
    int    nr;    /* System call number */
    __u32  arch;  /* AUDIT_ARCH_* value */
    __u64  instruction_pointer; /* CPU IP */
    __u64  args[6]; /* System call arguments */
};
```
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_I386`, `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
Obviously, one can code BPF instructions numerically by hand.

But, header files define symbolic constants and convenience macros \( \text{BPF_STMT()} \), \( \text{BPF_JUMP()} \) to ease the task.

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

(Macros just plug values together to form structure)
Building BPF instructions: examples

- Load architecture number into accumulator

\[
\text{BPF_STMT(} \text{BPF_LD} \mid \text{BPF_W} \mid \text{BPF_ABS},
\text{offsetof(} \text{struct seccomp_data, arch)}))
\]

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

- Operand is \text{architecture} field of data area
  - \text{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 with SIGSYS

```c
BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_KILL)
```
Checking the architecture

- Checking architecture value should be first step in any BPF program
- Architecture may support multiple system call conventions
  - E.g. x86 hardware supports x86-64 and i386
  - System call numbers may differ or overlap
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 mask) specify an action to kernel
  - Least significant 16 bits (SECCOMP_RET_DATA mask) specify “data” for return value
Filter return action component is one of

- **SECCOMP_RET_ALLOW**: system call is executed
- **SECCOMP_RET_KILL**: process is immediately terminated
  - Terminated as though process 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
- **SECCOMP_RET_TRACE**: attempt to notify ptrace() tracer
  - Gives tracing process a chance to assume control
  - See seccomp(2)
- **SECCOMP_RET_TRAP**: process is sent SIGSYS signal
  - Can catch this signal; see seccomp(2) for more details
Outline

1 Introduction and history
2 Seccomp filtering and BPF
3 Constructing seccomp filters
4 BPF programs
5 Further details on seccomp filters
6 Applications, tools, and further information
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 NS)
- Caller has to set the no_new_privs process attribute:

  \[
  \text{prctl(PR\_SET\_NO\_NEW\_PRIVS, 1, 0, 0, 0, 0)};
  \]

- 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
- \(! \text{no_new_privs} \&\& ! \text{CAP\_SYS\_ADMIN} \Rightarrow \text{seccomp()/prctl(PR\_SET\_SECCOMP)} \) fails with EACCES
Program installs a filter that prevents `open()` being called, and then calls `open()`

- Set `no_new_privs` bit
- Install seccomp filter
- Call `open()`
Example: seccomp/seccomp_deny_open.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),
    ...
  }

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

```c
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
    (offsetof(struct seccomp_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_RET | BPF_K, SECCOMP_RET_KILL)
```

- Remainder of filter program
- Load system call number into accumulator
- Test if system call number matches __NR_open
  - True: advance one instruction ⇒ kill process
  - False: advance 0 instructions ⇒ allow system call
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” 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()

- O_CREAT specified ⇒ kill process
- O_WRONLY or O_RDWR specified ⇒ cause open() to fail with ENOTSUP error
Example: seccomp/seccomp_control_open.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,
               AUDIT_ARCH_X86_64, 1, 0),
    BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_KILL),

    BPF_STMT (BPF_LD | BPF_W | BPF_ABS,
               (offsetof(struct seccomp_data, nr))),
    BPF_JUMP (BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 1, 0),
    BPF_STMT (BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
};
```

- Load architecture and test for expected value
- Load system call number
- Test if system call number is `__NR_open`
  - True: skip next instruction
  - False: skip 0 instructions ⇒ permit all other syscalls
Example: seccomp/seccomp_control_open.c

```c
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
          (offsetof(struct seccomp_data, args[1]))),
BPF_JUMP(BPF_JMP | BPF_JSET | BPF_K, O_CREAT, 0, 1),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
```

- Load second argument of `open()` (flags)
- Test if `O_CREAT` bit is set in flags
  - True: skip 0 instructions ⇒ kill process
  - False: skip 1 instruction
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_WRONLY or O_RDWR are set in flags
  - True: cause open() to fail with ENOTSUP error in errno
  - False: allow open() 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
$ ./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 and history
2. Seccomp filtering and BPF
3. Constructing seccomp filters
4. BPF programs
5. Further details on seccomp filters
6. Applications, tools, and further information
Installing multiple filters

- If existing filters permit `prctl()` or `seccomp()`, further filters can be installed.
- **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` (highest priority)
  - `SECCOMP_RET_TRAP`
  - `SECCOMP_RET_ERRNO`
  - `SECCOMP_RET_TRACE`
  - `SECCOMP_RET_ALLOW` (lowest priority)
fork() and execve() semantics

- If seccomp filters permit fork() or clone(), then child inherits parents 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)

- Obviously, order of filtering rules can affect performance
  - Construct filters so that most common cases yield shortest execution paths
  - If handling many different system calls, binary chop techniques can give $O(\log N)$ performance
Outline

1. Introduction and history
2. Seccomp filtering and BPF
3. Constructing seccomp filters
4. BPF programs
5. Further details on seccomp filters
6. Applications, tools, and further information
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
- Failure-mode testing
  - Place application in environment where unusual / unexpected failures occur
  - Blacklist certain syscalls / argument combinations to generate failures
Tools: *libseccomp*

- High-level API for kernel creating seccomp filters
  - [https://github.com/seccomp/libseccomp](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
  - Can output generated code in binary (for seccomp filtering) or human-readable form (“pseudofilter code”)
  - Don’t have full control of generated code, but can give hints about which system calls to prioritize in generated code
- [http://lwn.net/Articles/494252/](http://lwn.net/Articles/494252/)
- Fully documented with man pages that contain examples (!)
Other tools

- **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`
  - See `bpf(2)` man page
Resources

- Kernel source files:
  Documentation/prctl/seccomp_filter.txt,
  Documentation/networking/filter.txt

- http://outflux.net/teach-seccomp/
  Shows handy trick for discovering which of an application’s system calls don’t pass filtering

- 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!

mtk@man7.org    @mkerrisk
Slides at http://man7.org/conf/

Linux System Programming,
System Programming for Linux Containers,
and other training at http://man7.org/training/