Using seccomp to limit the kernel attack surface

Michael Kerrisk, man7.org © 2023

mtk@man7.org

19 January 2023, Oslo
# 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>8</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>An example</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Productivity aids (libseccomp and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Further details on seccomp filters</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Further information</td>
<td>60</td>
</tr>
</tbody>
</table>
Who?

- Linux *man-pages* project
    - Approx. 1060 pages documenting syscalls and C library
  - Contributor since 2000
  - Maintainer 2004-2020
  - Comaintainer 2020-2021

- I wrote a book
- Trainer/writer/engineer
  - http://man7.org/training/
- mtk@man7.org, @mkerrisk

©2023, Michael Kerrisk @mkerrisk
<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>8</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>An example</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Productivity aids (libseccomp and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Further details on seccomp filters</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Further information</td>
<td>60</td>
</tr>
</tbody>
</table>
What is seccomp?

- Kernel provides large number of system 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
- 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)
  - But, much simpler functionality
- 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, Firefox, OpenSSH, `vsftpd`, `systemd`, Docker, LXC, Flatpak, Firejail, `strace`
- Linux 3.17 (2014):
  - `seccomp()` system call added
    - Provides additional seccomp functionality that is unavailable via `prct()`
- And work is ongoing...
1 Introduction
2 Introduction to Seccomp
3 Seccomp filtering and BPF
4 The BPF virtual machine and BPF instructions
5 BPF filter return values
6 Installing BPF programs
7 An example
8 A more sophisticated example
9 Checking the architecture
10 Productivity aids (libseccomp and other tools)
11 Further details on seccomp filters
12 Caveats
13 Further information
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
  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 byte code

- Seccomp filters are expressed as BPF (Berkeley Packet Filter) programs
- BPF is a **byte code which is interpreted by a virtual machine (VM) implemented inside kernel**
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 packets are discarded early
- **Avoid expense of passing every** packet over kernel-user-space boundary
- ☺️ Seccomp ⇒ 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)
  - Kernel can 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
1 Introduction 3
2 Introduction to Seccomp 5
3 Seccomp filtering and BPF 8
4 The BPF virtual machine and BPF instructions 13
5 BPF filter return values 24
6 Installing BPF programs 27
7 An example 30
8 A more sophisticated example 36
9 Checking the architecture 43
10 Productivity aids (libseccomp and other tools) 47
11 Further details on seccomp filters 52
12 Caveats 56
13 Further information 60
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>`

- **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)
- 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 $\Rightarrow$ 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

```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 */
};
```

- **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

- 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 `sock_filter` structure initializer

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

\[
\text{BPF\_JUMP}((\text{BPF\_JMP} \mid \text{BPF\_JEQ}) \mid \text{BPF\_K}, \text{AUDIT\_ARCH\_X86\_64}, 1, 0)
\]

- **BPF\_JMP | BPF\_JEQ**: jump with test on equality
- **BPF\_K**: value to test against is in generic multiuse field \((k)\)
- \(k\) contains value \text{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**
  
  ```
  BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
  ```

- **Arithmetic/logic instruction: add one to accumulator**
  
  ```
  BPF_STMT(BPF_ALU | BPF_ADD | BPF_K, 1)
  ```

- **Arithmetic/logic instruction: right shift accumulator 12 bits**
  
  ```
  BPF_STMT(BPF_ALU | BPF_RSH | BPF_K, 12)
  ```
<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>8</td>
</tr>
<tr>
<td>4. The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5. BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6. Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7. An example</td>
<td>30</td>
</tr>
<tr>
<td>8. A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9. Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10. Productivity aids (libseccomp and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11. Further details on seccomp filters</td>
<td>52</td>
</tr>
<tr>
<td>12. Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13. Further information</td>
<td>60</td>
</tr>
</tbody>
</table>
Filter return value

- Once a filter is installed, each system call is tested against filter.
- Seccomp filter returns a value to kernel indicating whether system call is permitted.
- 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

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**, **SECCOMP_RET_USER_NOTIF**
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:

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

- 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)
Example: seccomp/seccomp_deny_open.c

```c
#define __NR_open /* Not all architectures have open() */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
#endif
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)
```

- **Test if system call number matches __NR_open**
  - True: advance 2 instructions ⇒ kill process
  - False: advance 0 instructions ⇒ next test
  - *(open()) is absent on some architectures, because it can be implemented using openat()*)

- **Test if system call number matches __NR_openat**
  - True: advance 1 instruction ⇒ kill process
  - False: advance 0 instructions ⇒ allow syscall
Example: seccomp/seccomp_deny_open.c

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

seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog);
```

- Construct argument for `seccomp()`
- Install filter
Example: `seccomp/seccomp_deny_open.c`

Upon running the program, we see:

```
$ ./seccomp_deny_open
Bad system call    # Message printed by shell
```

- “Bad system call” was printed by shell, because it looks like its child was killed by `SIGSYS`
<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>8</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>An example</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Productivity aids (<a href="#">libseccomp</a> and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>Further details on seccomp filters</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Further information</td>
<td>60</td>
</tr>
</tbody>
</table>
A more sophisticated example

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

- *O_CREAT* specified ⇒ kill process
- *O_WRONLY* or *O_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
struct sock_filter filter[] = {
  /* Architecture-check code not shown */

  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
           (offsetof(struct seccomp_data, nr))),
...

#ifdef __NR_open /* Not all architectures have open() */
  /* Is this an open() syscall? */
  BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 0, 2),
  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
           (offsetof(struct seccomp_data, args[1]))),
  BPF_JUMP(BPF_JMP | BPF_JA, 3, 0, 0),
#endif
```

- Load system call number
- For `open()`, load `flags` argument (`args[1]`) into accumulator, and then skip to `flags` processing
  - (Some architectures don't have `open()`)
Example: seccomp/seccomp_control_open.c

```c
/* Is this an openat() syscall? */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 1, 0),

BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),

BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
         (offsetof(struct seccomp_data, args[2]))),
```

- For `openat()`, load `flags` argument (`args[2]`) into accumulator and continue to `flags` processing
- Allow all other system calls
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)
```
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
$ touch /tmp/a
$ ./seccomp_control_open
open2: Operation not supported
open3: Operation not supported
Bad system call

- First *open()* succeeded
- Second and third *open()* calls failed
  - Kernel produced **ENOTSUP** error for call
- Fourth *open()* call caused process to be killed
<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>8</td>
</tr>
<tr>
<td>4 The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5 BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6 Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7 An example</td>
<td>30</td>
</tr>
<tr>
<td>8 A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9 Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10 Productivity aids (libseccomp and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11 Further details on seccomp filters</td>
<td>52</td>
</tr>
<tr>
<td>12 Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13 Further information</td>
<td>60</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
  - Modern x86 hardware supports three(!) architecture+ABI conventions
  - System call numbers may differ under each convention
  - Similar issues occur on other platforms
    - E.g., AArch64 can execute AArch32 code, but set of syscalls differs somewhat on each architecture
Checking the architecture: Intel architectures

- E.g. modern Intel systems support x86-64, i386, and x32, each of which has unique syscall numbers
  - x86-64 (**AUDIT_ARCH_X86_64**): modern x86 arch. with 64-bit instructions, larger address space, richer register set
  - i386 (**AUDIT_ARCH_I386**): historical 32-bit Intel arch. with 32-bit instruction set and address space
  - x32 ABI (Linux 3.4, 2012): use modern x86 arch. with 32-bit pointers/**long**
    - Can result in more compact/faster code in some cases
    - ⚠️ Same **arch** value (**AUDIT_ARCH_X86_64**) as x86-64, but bit 30 (**X32_SYSCALL_BIT**) set in syscall number (**nr**) is set in syscall number (**nr**) set in syscall number (**nr**)
  - Checking **arch** in each filter invocation is **essential** because architecture may change over life of process (**execve()**)
# define X32_SYSCALL_BIT 0x40000000
...

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, 0, 2),

    BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
             (offsetof(struct seccomp_data, nr))),
    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, X32_SYSCALL_BIT, 0, 1),

    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
}

- Load architecture; kill process if not as expected
- Load system call number; kill process if this is an x32 system call (bit 30 is set)
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
  - 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/](http://lwn.net/Articles/494252/)
- Fully documented with manual pages containing examples(!)
libseccomp example (seccomp/libseccomp_demo.c)

```c
scmp_filter_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); /* Load filter */
seccomp_release(ctx); /* Free filter state */

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, and free user-space filter state (no longer needed)
- Try calling `fork()`
Example run (seccomp/libseccomp_demo.c)

$ ./libseccomp_demo
fork: Operation not permitted

- *fork()* fails, as expected
- EPERM error $\Rightarrow$ *fork()* wrapper in glibc calls *clone()* (!)
Other productivity aids

- **easyseccomp** - a DSL for writing seccomp filters
  - [https://github.com/giuseppe/easyseccomp](https://github.com/giuseppe/easyseccomp)
  - New in 2021; worth watching, to see future progress

- **bpfc** (BPF compiler)
  - Compiles assembler-like BPF programs to byte code
  - Part of *netsniff-ng* project ([http://netsniff-ng.org/](http://netsniff-ng.org/))
<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
<th></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>8</td>
</tr>
<tr>
<td>4</td>
<td>The BPF virtual machine and BPF instructions</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>BPF filter return values</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Installing BPF programs</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>An example</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>A more sophisticated example</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Checking the architecture</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Productivity aids (<em>libseccomp</em> and other tools)</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td><strong>Further details on seccomp filters</strong></td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>Caveats</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Further information</td>
<td>60</td>
</tr>
</tbody>
</table>
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()*
  - seccomp/seccomp_launch.c: launch a program after first loading a specified BPF blob from a file
Cost of filtering, construction of filters

- Installed BPF filter(s) are executed for every system call
  - \( \Rightarrow \) there's a performance cost

**Indicative** timings on x86-64, Linux 5.2:

- `seccomp/seccomp_perf.c`
  - Performs 6 BPF instructions / permitted syscall
  - Call `getppid()` repeatedly (one of cheapest syscalls)
  - +20% (JIT compiler enabled); +75% execution time (JIT compiler disabled)
  - Looks relatively high because `getppid()` is a cheap syscall
Cost of filtering, construction of filters

- Obviously, order of filtering rules can affect performance
  - ⇒ construct filters so that most common cases yield shortest execution paths

- But: a significant part of cost seems to be filter start-up / termination
  - Even a filter consisting of just one (return) instruction adds 10% to `getppid()` loop
  - And different BPF instructions (unsurprisingly) have different costs
  - See `seccomp/seccomp_bench.c`
There are subtleties when it comes to deploying seccomp filters:

- Adding a seccomp filter can **cause** bugs in application:
  - What if filter disallows a system call that should have been allowed?
    - ⇒ A buggy filter might **cause a legitimate application action to fail**
  - Such bugs 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)
  - Following slides...
Caveats

- Filtering is based on syscall numbers, but applications normally call C library wrappers; some implications:
  - Some wrapper functions use syscalls of a different name
    - Must filter for the correct underlying syscall
    - E.g., glibc `fork()` wrapper actually calls `clone()`
  - Wrapper function behavior may change across glibc versions
    - E.g., in glibc 2.26, the `open()` wrapper switched from using `open(2)` to using `openat(2)`
    - Such changes in the C library are ongoing (and necessary)
    - A robust filter will filter all related system calls
  - Wrapper function behavior may vary across C libraries
    - E.g., musl libc vs glibc
Caveats

- Moral of the story: BPF filters are like any other production code
  - They need unit tests
  - They need CI testing
  - They need to be tested on all platforms and architectures where they might be deployed
  - This is far from easy...
    - A war story: https://github.com/kristapsdz/acme-client-portable/blob/master/Linux-seccomp.md
Resources

- Kernel source files:
  - Documentation/userspace-api/seccomp_filter.rst
  - Documentation/networking/filter.txt BPF VM in detail

- [http://outflux.net/teach-seccomp/](http://outflux.net/teach-seccomp/)

- `seccomp(2)` man page

- “Seccomp sandboxes and memcached example”
  - [https://blog.viraptor.info/post/seccomp-sandboxes-and-memcached-example-part-1](https://blog.viraptor.info/post/seccomp-sandboxes-and-memcached-example-part-1)

- [https://lwn.net/Articles/656307/](https://lwn.net/Articles/656307/)
  - Write-up of a version of this presentation...
Thanks!

Michael Kerrisk, Trainer and Consultant
http://man7.org/training/

mtk@man7.org    @mkerrisk

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