

Seccomp

Michael Kerrisk, man7.org © 2026

mtk@man7.org

January 2026

Outline

Rev: #d49d0b54be95

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

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 system calls
 - Remaining systems calls should never legitimately occur
 - If they do occur, perhaps it is because program has been compromised
- Seccomp (“**secure computing**”) = mechanism to restrict system calls that a process may make
 - Reduces attack surface of kernel
 - A key component for building application sandboxes
- Used by many apps; e.g., Chrome, Firefox, OpenSSH, *vsftpd*, *systemd*, Docker, LXC, Flatpak, *strace*



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Seccomp filtering

- Allows filtering based on system call number and argument (register) values
 - Pointers can **not** be dereferenced
 - Because of time-of-check, time-of-use race conditions
Seccomp and deep argument inspection
<https://lwn.net/Articles/822256/>, June 2020
 - Landlock LSM, added in Linux 5.13 (2021), addresses this restriction(?)

Seccomp filtering overview

- Steps:
 - 1 Construct filter program that specifies permitted syscalls
 - 2 Process installs filter into kernel
 - 3 Process executes code that should be filtered
 - For example: `exec()` new program, or invoke function in dynamically loaded library (plug-in)
- Once installed, **every syscall made by process triggers execution of filter**
- Installed filters **can't** be removed
 - Filter == declaration that we don't trust subsequently executed code
 - Filters are inherited by children of `fork()`
 - Filters are preserved during `exec()`



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
 - <http://www.tcpdump.org/papers/bpf-usenix93.pdf>
- 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



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



- 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
 - Programs are limited to 4096 instructions
 - Only **branch-forward** instructions
 - Programs are directed acyclic graphs (DAGs)
 - Kernel can verify validity/safety of programs
 - Program completion is guaranteed (DAGs)
 - Simple instruction set \Rightarrow can verify opcodes and arguments
 - Can detect dead code



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

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:

```
struct sock_filter {  
    __u16 code;      /* Filter code (opcode) */  
    __u8 jt;         /* Jump true */  
    __u8jf;         /* 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 \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* 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 (4 bytes) */  
    __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 convenience macros (and symbolic constants) 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 }
```

- These macros just plug values together to form *sock_filter* structure initializer

```
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

```
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

```
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 a 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)
```



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Filter return value

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

```
#define SECCOMP_RET_ACTION_FULL 0xffff0000U
#define SECCOMP_RET_DATA        0x0000ffffU
```

Possible filter return actions include:

- `SECCOMP_RET_ALLOW`: system call is allowed to execute
- `SECCOMP_RET_KILL_PROCESS` (since Linux 4.14, 2017): 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_ERRNO`: return an error from system call
 - System call is not executed
 - Value in `SECCOMP_RET_DATA` is returned in `errno`
 - But, capped to 4095
- There are other possible return actions....
 - See `seccomp(2)` manual page



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<code>libseccomp</code> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Installing a BPF program

- A process installs a filter for itself using one of:
 - `seccomp(SECCOMP_SET_MODE_FILTER, flags, &fprog)`
 - Since Linux 3.17 (2014)
 - Provides additional features unavailable with `prctl()`
 - `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &fprog)`
 - Legacy mechanism for installing seccomp filter
- `&fprog` is a pointer to a BPF program:

```
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 (`CAP_SYS_ADMIN` in its user namespace)
- Caller has set the `no_new_privs` process attribute:

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

- Per-thread attribute; once set, `no_new_privs` can't be unset
- `! no_new_privs && ! CAP_SYS_ADMIN` ⇒ `seccomp()`/`prctl(PR_SET_SECCOMP)` fails with `EACCES`



Effect of `no_new_privs` process attribute

Effect of `no_new_privs` process attribute:

- In subsequent `execve()` calls:
 - Causes set-UID and set-GID bits to be ignored
 - Prevents file from conferring permitted capabilities to process[†]
- Prevents possibility of attacker starting privileged program and manipulating it to misbehave using a seccomp filter

[†]More precisely: after `execve()`, process permitted set will be intersection of [process permitted set before `execve()`] and [file permitted set]



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Example: seccomp/seccomp_deny_open.c

```
1 int main(int argc, char *argv[]) {
2     prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
3
4     install_filter();
5
6     open("/tmp/a", O_RDONLY);
7
8     printf("We shouldn't see this message\n");
9     exit(EXIT_SUCCESS);
10 }
```

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

```
1 static void install_filter(void) {
2     struct sock_filter filter[] = {
3
4         /* Architecture-check code not shown */
5
6         BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
7                 offsetof(struct seccomp_data, nr)),
8         ...
9     };
10 }
```

- 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

```
1 #ifdef __NR_open      /* Not all architectures have open() */
2     BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
3 #endif
4     BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 1, 0),
5     BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
6     BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
7 }
```

- Test if system call number matches `__NR_open`
 - True: advance 2 instructions \Rightarrow kill process
 - False: advance 0 instructions \Rightarrow 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 \Rightarrow kill process
 - False: advance 0 instructions \Rightarrow allow syscall



man7.org

Example: seccomp/seccomp_deny_open.c

```
1     struct sock_fprog prog = {
2         .len = sizeof(filter) / sizeof(filter[0]),
3         .filter = filter,
4     };
5
6     if (seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog) == -1)
7         errExit("seccomp");
8 }
```

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



man7.org

Example: seccomp/seccomp_deny_open.c

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” was 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
 - (List signals and their numbers with: `kill -l`)



Example: seccomp/seccomp_control_open.c

- A more sophisticated example
- Filter based on *flags* argument of `open()` / `openat()`
 - `O_CREAT` specified \Rightarrow kill process
 - `O_WRONLY` or `O_RDWR` specified \Rightarrow cause call to fail with `ENOTSUP` error
- flags* is arg. 2 of `open()`, and arg. 3 of `openat()`:

```
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

```
struct sock_filter filter[] = {
    /* Architecture-check code not shown */

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

```
/* 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)  
};
```

Process *flags* value:

- 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

```
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

```
$ touch /tmp/a
$ ./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
 - (159 == 128 + 31; `SIGSYS` is signal 31)



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

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 (Linux 3.4, 2012): use modern x86 arch. with 32-bit pointers/*long* (not widely supported in distributions)
 - 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*)
- Checking *arch* in each filter invocation is **essential** because **architecture may change over life of process** (`execve()`)
 - Interesting experiment in `seccomp/seccomp_multiarch.c`



Checking the architecture: Intel x86-64

```
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)),
#define X32_SYSCALL_BIT          0x40000000
    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, X32_SYSCALL_BIT, 0, 1),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
    /* Further BPF code... */
```

- 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)



Checking the architecture: AArch64

```
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_AARCH64, 1, 0),
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
    /* Further BPF code... */
```

- Load architecture; kill process if not as expected



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<i>libseccomp</i> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68

Notes for online practical sessions

- Small groups in **breakout rooms**
 - Write a note into the Discord `#general` channel if you have a preferred group
- **We will go faster, if groups collaborate** on solving the exercise(s)
 - You can **share a screen** in your room
- I will circulate regularly between rooms to answer questions
- Zoom has an “**Ask for help**” button...
- **Keep an eye on the Discord `#general` channel**
 - Perhaps with further info about exercise;
 - Or a note that the exercise merges into a break
- When your room has finished, write a message in the Discord `#general` channel: “***** Room X has finished *****”
 - Then I have an idea of how many people have finished

Shared screen etiquette

- It may help your colleagues if you **use a larger than normal font!**
 - In many environments (e.g., *xterm*, *VS Code*), we can adjust the font size with **Control+Shift+“+”** and **Control+“-”**
 - Or (e.g., *emacs*) hold down **Control** key and use mouse wheel
- **Long shell prompts** make reading your shell session difficult
 - Use `PS1='\$ '` or `PS1='# '`
- **Low contrast** color themes are difficult to read; change this if you can
- Turn on **line numbering** in your editor
 - In *vim* / *neovim* use: `:set number`
 - In *emacs* use: `M-x display-line-numbers-mode <RETURN>`
 - `M-x` means **Left-Alt+x**
- For collaborative editing, **relative line-numbering is evil....**
 - In *vim* / *neovim* use: `:set nornu`
 - In *emacs*, the following should suffice:

`M-: (display-line-numbers-mode) <RETURN>`
`M-: (setq display-line-numbers 'absolute) <RETURN>`



Using *tmate* in in-person practical sessions

In order to share an X-term session with others, do the following:

- Enter the command *tmate* in an X-term, and you'll see the following:

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io
web session: ...
ssh session: ssh S0mEoTheRrAnD0m5Tr1Ng@lon1.tmate.io
```

- Share last “ssh” string with colleague(s) via a text channel
 - Or: “ssh session read only” string gives others read-only access
- Your colleagues should paste that string into an X-term...
- Now, you are sharing an X-term session in which anyone can type
 - Any “mate” can cut the connection to the session with the 3-character sequence `<ENTER> ~`.
- To see above message again: `tmate show-messages`



Exercises

- 1 Extend the filter in `seccomp/seccomp_control_open.c` so that if `O_TRUNC` is specified in the `flags` argument, the call fails with the error `EACCES`, and adjust `main()` to test for this case.
- 2 Write a program ([template: `seccomp/ex.seccomp_no_children.c`], and you will need to edit the `EXERCISE_FILES_EXE` line in the `Makefile` so that it builds this program) that installs a filter that denies execution of `fork()`, `clone()`, and `clone3()`, causing `fork()` to fail with `ENOTSUP`, `clone()` to fail with `EPERM`, and `clone3()` to fail with `EACCES`. (Note: there is no `fork()` system call on certain architectures, such as AArch64, so you should not include try to filter for `fork()` calls on those architectures.) The program should support the following command-line arguments:

```
./ex.seccomp_no_children prog arg...
```

Having installed the filter, the program should then `exec()` `prog` with the supplied arguments. Obviously, an interesting program to `exec()` is one that employs `fork()` or `clone()`. Try executing the `proceexec/zombie.c` program (which calls `fork()`). What error does the program fail with? (Read VERSIONS (or NOTES) in `fork(2)` to understand why.)

- 3 ⚡ ⚡ ⚡ Extend the `seccomp/seccomp_deny_open.c` program so that it also denies the `creat()`, `openat2()`, and `open_by_handle_at()` system calls.



Outline

24 Seccomp	24-1
24.1 Introduction	24-3
24.2 Seccomp filtering and BPF	24-5
24.3 The BPF virtual machine and BPF instructions	24-12
24.4 BPF filter return values	24-23
24.5 Installing a BPF program	24-26
24.6 BPF program examples	24-30
24.7 Checking the architecture	24-42
24.8 Exercises	24-47
24.9 Productivity aids (<code>libseccomp</code> and other tools)	24-52
24.10 Performance considerations	24-62
24.11 Homework exercises	24-65
24.12 Applications and further information	24-68