Outline

6 Seccomp 6-1
6.1 Introduction and history 6-3
6.2 Seccomp filtering and BPF 6-11
6.3 Constructing seccomp filters 6-18
6.4 BPF programs 6-33
6.5 Recent seccomp features 6-49
6.6 Discovering the system calls made by a program 6-52
6.7 Further details on seccomp filters 6-59
6.8 Applications, tools, and further information 6-65
What is seccomp?

- Kernel provides large number of systems calls
  - ≈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
### 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, Flatpak, Firejail
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>Value</th>
<th>Description</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...

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)
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()
  - Construct and install BPF filter
  - 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 ⇒ can verify opcodes and arguments
    - Can detect dead code
    - Can verify that program completes via a “return” instruction
    - BPF filter programs are limited to 4096 instructions
Generalizing BPF

- BPF originally designed to work with network packet headers
- 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

Generalizing BPF: eBPF

- An extended BPF language (eBPF) has been developed
  - To make distinction clear, original BPF is sometimes call classic BPF (cBPF)
  - Notable features of eBPF:
    - Somewhat larger instruction set
    - Ability to call a specified set of in-kernel helper functions
    - Maps (persistent associative arrays that can be shared between BPF programs and with user-space)
  - See bpf(2)
  - Intended that one day seccomp() filter programs can use eBPF
Generalizing BPF: eBPF

- eBPF is appearing in very many use cases:
  - Linux 3.18: adding eBPF filters to kernel tracepoints
  - Linux 3.19: adding eBPF filters to raw sockets
  - Linux 4.4: eBPF filtering of perf events
  - Linux 4.5: use eBPF program to distribute packets to SO_REUSEPORT group of sockets
  - Linux 4.7: attach eBPF filters to perf tracepoints
6 Seccomp
6.1 Introduction and history 6-3
6.2 Seccomp filtering and BPF 6-11
6.3 Constructing seccomp filters 6-18
6.4 BPF programs 6-33
6.5 Recent seccomp features 6-49
6.6 Discovering the system calls made by a program 6-52
6.7 Further details on seccomp filters 6-59
6.8 Applications, tools, and further information 6-65

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:

```
struct sock_filter {
    __u16 code;    /* Filter block */
    __u8  jt;      /* Filter code (opcode) */
    __u8  jf;      /* Jump true */
    __u32  k;      /* Jump false */
    /* 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

```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)
- **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

Building BPF instructions

- Obviously, one can code BPF instructions numerically by hand
- But, header files define symbolic constants and convenience macros (`BPF_STMT()`, `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
  
  ```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
  - **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

- 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_FULL mask) specify an action to kernel
  - Least significant 16 bits (SECCOMP_RET_DATA mask) specify “data” for return value
Filter return action (1)

Filter return action component is one of:

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

Filter return action (2)

- **SECCOMP_RET_ERRNO**: return an error from system call
  - System call is not executed
  - Value in **SECCOMP_RET_DATA** is returned in `errno`
- **SECCOMP_RET_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
- **SECCOMP_RET_LOG** (since Linux 4.14): allow + log syscall
  - System call is allowed, and also logged to audit log
    - `/var/log/audit/audit.log`; `ausearch(8)`
  - Useful during filter development (later...)
Installing a BPF program

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

- `&fprog` is a pointer to a BPF program:

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

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

- Caller is privileged (has `CAP_SYS_ADMIN` in its user NS)
- 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
- Per-thread attribute
- Prevents possibility of attacker starting privileged program and manipulating it to misbehave using a seccomp filter
- `!no_new_privs && !CAP_SYS_ADMIN` ⇒ `seccomp() / prctl(PR_SET_SECCOMP)` fails with `EACCES`

Example: seccomp/seccomp_deny_open.c

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

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

- Set `no_new_privs` bit
- Install seccomp filter
- Call `open()`
static void install_filter(void) {
    struct sock_filter filter[] = {
        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, 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)
    
    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 instr.)
      False: skip no instructions
    Kill process on architecture mismatch
    (BPF program continues on next slide)

    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

}
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() / openat()
  - O_CREAT specified ⇒ kill process
  - O_WONLY or O_RDWR specified ⇒ cause call to fail with ENOTSUP error

```
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))),
}
```

- Load architecture and test for expected value
- Load system call number
(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

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
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.

   Hints:
   - If using Linux 3.16 and earlier (or a system with older glibc headers that don’t define `__NR_seccomp`), the `seccomp()` call will need to be replaced with the use of `prctl(PR_SET_SECCOMP)` (see comments in the source code).
   - To successfully `make(1)`, you may need to install the `libseccomp-dev` (Debian) or `libseccomp-devel` (RPM) package.

2. Write a program ([template: `seccomp/ex.seccomp_no_children.c`]) that installs a filter that denies execution of `fork()` and `clone()`, causing `fork()` to fail with `ENOTSUP` and `clone()` to fail with `EPERM`. 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 `procexec/zombie.c` program (which calls `fork()`); what error does the program fail with?

3. Modify the `seccomp/seccomp_deny_open.c` program so that, after installing the BPF program, it prints its PID and uses the `sleep()` library function to sleep for 60 seconds before the call to `open()`. Run the program, and while it is sleeping, check the value of the `Seccomp` field in the corresponding `/proc/PID/status` file.