Secure Programming (06-20010)
Chapter 5: Unix Access Control Mechanisms

Christophe Petit

University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

Access control

Outline

Outline

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

Kernel space and User space

- Virtual memory divided into kernel space and user space
- Kernel space is for privileged operating system kernel, kernel extensions, and most device drivers
- User space is where applications and some drivers execute
- This protects memory and hardware from both malicious and buggy software
- System calls allow users to call kernel operations

Users and Groups

- Users are identified by their User ID (UID)
- UID 0 is a special privilege user called root, typically the administrator
- Unprivileged user IDs typically start at 500 or 1000

- A group is a set of users sharing resources (files, devices, programs)
- Each group has a Group ID (GID)
- Each user belongs to at least one group, and potentially supplementary groups

etc/passwd and etc/shadow

- etc/passwd contains the list of users
- Can be read by any user
- Contains lines such as

  chris : x : 500 : 500 : Christophe Petit : /home/chris : /bin/bash
- etc/shadow contains (salted) password hashes
- Can only be read by root

Filesystem objects

- Information organized in a directory tree rooted at "/", where each directory contains filesystem objects
- Filesystem objects can be ordinary files, directories, symbolic links, named pipes, sockets,...

Usual directories

- /etc : configuration files
- /home : user files and applications
- /bin : executables that are part of the OS
- /sbin : executables for superusers
- /var : log files, temporary files

Filesystem object attributes

- Owning UID and GUID
  - Can only be changed by owner and root
- Permission bits : read, write and execute permissions for owner, group and other
  - Permission to add/remove files depends on the file's directory attributes, not the file's attributes
- Sticky bit : on a directory, prevents removal and renames on its files (except by file owner, directory owner and root)
- setuid, setgid : when set on executable file, program runs with privileges of the file owner instead of executer
- Timestamps storing access and modification times

File permissions

- Each file has attached read - write - execute permissions for owner - group -other
- Example : permission 754 means
  - File owner can read, write, execute ($7 = 4 \cdot 1 + 2 \cdot 1 + 1 \cdot 1$)
  - Group owner can read, not write, execute ($5 = 4 \cdot 1 + 2 \cdot 0 + 1 \cdot 1$)
  - Others can read, not write, not execute($4 = 4 \cdot 1 + 2 \cdot 0 + 1 \cdot 0$)
- Note : with 457 file owner can only read
- For directories, permission bits mean listing files, adding/removing/renaming files, and access all files

Changing Access Control Attributes

chmod(1) - Linux man page

Name

chmod - change file mode bits

Synopsis

chmod [*OPTION*]... *MODE*[*,MODE*]... *FILE*...
chmod [*OPTION*]... *OCTAL-MODE FILE*...
chmod [*OPTION*]... *--reference=RFILE FILE*...

Description

This manual page documents the GNU version of **chmod**. **chmod** changes the file mode bits of each given file according to *mode*, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

The format of a symbolic mode is [**ugoa**...][[**+-=**][*perms*...]...], where *perms* is either zero or more letters from the set **rwxXst**, or a single letter from the set **ugo**. Multiple symbolic modes can be given, separated by commas.

A combination of the letters **ugoa** controls which users' access to the file will be changed: the user who owns it (**u**), other users in the file's group (**g**), other users not in the file's *group (**o**)*, or all users (**a**). If none of these are given, the effect is as if **a** were given, but bits that are set in the umask are not affected.

The operator **+** causes the selected file mode bits to be added to the existing file mode bits of each file; **-** causes them to be removed; and **=** causes them to be added and causes unmentioned bits to be removed except that a directory's unmentioned set user and group ID bits are not affected.

The letters **rwxXst** select file mode bits for the affected users: read (**r**), write (**w**), execute (or search for directories) (**x**), execute/search only if the file is a directory or already has execute permission for some user (**X**), set user or group ID on execution (**s**), restricted deletion flag or sticky bit (**t**). Instead of one or more of these letters, you can specify exactly one of the letters **ugo**: the permissions granted to the user who owns the file (**u**), the permissions granted to other users who are members of the file's group (**g**), and the permissions granted to users that are in neither of the two preceding *categories (**o**)*.

- See also fchmod, chown, chgrp

Use of Access Control Attributes

- Checked when opening a file
- Not checked at every read/write
- Checked by unix functions open, creat, link, unlink, rename, mknod, symlink, socket

Symbolic links (symlinks)

- Symlinks are references to other files
- Automatically resolved by the operating system
- Every user on the local system can create symlinks
  - Link target does not need to be owned by user
  - User needs write permission on the directory where they create the symlink

ln command

LN(1) User Commands LN(1)

NAME top

ln - make links between files

SYNOPSIS top

```
ln [OPTION]... [-T] TARGET LINK_NAME   (1st form)
ln [OPTION]... TARGET                  (2nd form)
ln [OPTION]... TARGET... DIRECTORY     (3rd form)
ln [OPTION]... -t DIRECTORY TARGET...  (4th form)
```

DESCRIPTION top

In the 1st form, create a link to TARGET with the name LINK_NAME. In the 2nd form, create a link to TARGET in the current directory. In the 3rd and 4th forms, create links to each TARGET in DIRECTORY. Create hard links by default, symbolic links with **--symbolic**. By default, each destination (name of new link) should not already exist. When creating hard links, each TARGET must exist. Symbolic links can hold arbitrary text; if later resolved, a relative link is interpreted in relation to its parent directory.

Processes

- User-level activities implemented by running processes
- Processes can create other processes with *fork*
- In Linux, *clone* can decide what resources are shared with process created

fork and clone

```
FORK(2)                Linux Programmer's Manual                FORK(2)

NAME  top

       fork - create a child process

SYNOPSIS  top

       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION  top

       fork() creates a new process by duplicating the calling process.  The
       new process is referred to as the child process.  The calling process
       is referred to as the parent process.

       The child process and the parent process run in separate memory
       spaces.  At the time of fork() both memory spaces have the same
       content.  Memory writes, file mappings (mmap(2)), and unmappings
       (munmap(2)) performed by one of the processes do not affect the
       other.

       The child process is an exact duplicate of the parent process except
       for the following points:

       *  The child has its own unique process ID, and this PID does not
          match the ID of any existing process group (setpgid(2)) or
          session.

       *  The child's parent process ID is the same as the parent's process
          ID.

       *  The child does not inherit its parent's memory locks (mlock(2),
          mlockall(2)).
```

```
CLONE(2)               Linux Programmer's Manual               CLONE(2)

NAME  top

       clone, __clone2 - create a child process

SYNOPSIS  top

       /* Prototype for the glibc wrapper function */

       #define _GNU_SOURCE
       #include <sched.h>

       int clone(int (*fn)(void *), void *child_stack,
                 int flags, void *arg, ...
                 /* pid_t *ptid, void *newtls, pid_t *ctid */ );

       /* For the prototype of the raw system call, see NOTES */

DESCRIPTION  top

       clone() creates a new process, in a manner similar to fork(2).

       This page describes both the glibc clone() wrapper function and the
       underlying system call on which it is based.  The main text describes
       the wrapper function; the differences for the raw system call are
       described toward the end of this page.

       Unlike fork(2), clone() allows the child process to share parts of
       its execution context with the calling process, such as the memory
       space, the table of file descriptors, and the table of signal
       handlers.  (Note that on this manual page, "calling process" normally
       corresponds to "parent process".  But see the description of
       CLONE_PARENT below.)
```

Process attributes

- Real-effective-saved user-group ID
- umask
- Resource limits

Real and effective IDs

- **Real ID** is ID of User executing the process
- **Effective ID** is used for most access checks
- Effective ID also determines owner of files created by the process

- By default Effective ID is Real ID
- Sometimes you need to use another user's identity, typically root, for privileged operations
- Setuid programs : Effective ID is ID owning the file, as opposed to ID running it

Saved ID

- **Saved ID** used for temporarily dropping permissions
  - Store effective ID in saved ID
  - Change effective ID to real ID
  - Later change effective ID back to saved ID
- An unprivileged process can only change its effective ID to saved ID or real ID
- There are also group versions of real-effective-saved IDs

Permissions for new files

- Every process has *umask* bit attributes
- System calls for file creation take a *mode* parameter, corresponding to read-write-execute permissions
- The *umask* bits tell which permissions must be *denied* on the new file, regardless of the system call argument
- Resulting file permissions are (!*umask*)&*mode*
- Example : if umask=022 and mode=777 we get 755

Example : etc/shadow

- etc/shadow typically contains hashes of user passwords
- File only accessible by root

- What if code on next slide executed by a normal user ?
- What if code on next slide executed by root ?

Opening etc/shadow

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void testFile() {
        char * filename = "/etc/shadow";
        FILE * f;
        f = fopen(filename, "r");
        if (f == NULL) {
                printf("failed!\n");
        }
        else {
                fclose(f);
                printf("OK\n");
        }
}

int main(int argc, char ** argv) {
        testFile();
}
```

Signals

- Interruption mechanism between processes
- On receiving signal the interrupted process must stop and handle it
- Examples are SIGSTOP, SIGCONT, SIGKILL
- Sending signals allowed when
  - Sending process is root
  - Real/effective UID of sending and receiving process equal
  - Special circunstances

Outline

setuid programs

- Motivation : allow ordinary users to perform functions which they could not perform otherwise
  - Allow users to see all active processes on a system

SYNOPSIS top

```
top -hv|-bcEHiOSs1 -d secs -n max -u|U user -p pid -o fld -w [cols]

The traditional switches `-' and whitespace are optional.
```

DESCRIPTION top

```
The top program provides a dynamic real-time view of a running
system.  It can display system summary information as well as a list
of processes or threads currently being managed by the Linux kernel.
The types of system summary information shown and the types, order
and size of information displayed for processes are all user
configurable and that configuration can be made persistent across
restarts.
```

  - Game score file

setuid programs : implementation

- Use setuid, setgid, seteuid, setegid functions to modify
  - Program file attributes setuid, setgid
  - Process attributes real/effective/saved user/group IDs
- setuid vs seteuid : when going from root to unprivileged, cannot go back to root if using setuid
- See also setreuid
- Should be use with care !
  " *explicitly violate UNIX protection mechanisms*"

setuid

SETUID(2) Linux Programmer's Manual SETUID(2)

NAME top

setuid - set user identity

SYNOPSIS top

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
```

DESCRIPTION top

setuid() sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the **CAP_SETUID** capability in its user namespace), the real UID and saved set-user-ID are also set.

Under Linux, **setuid**() is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some unprivileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken. The **setuid**() function checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to *uid*. After this has occurred, it is impossible for the program to regain root privileges.

seteuid

```
SETEUID(2)              Linux Programmer's Manual              SETEUID(2)
```

NAME top

```
       seteuid, setegid - set effective user or group ID
```

SYNOPSIS top

```
       #include <sys/types.h>
       #include <unistd.h>

       int seteuid(uid_t euid);
       int setegid(gid_t egid);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       seteuid(), setegid():
           _POSIX_C_SOURCE >= 200112L
               || /* Glibc versions <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION top

```
       seteuid() sets the effective user ID of the calling process.
       Unprivileged processes may only set the effective user ID to the real
       user ID, the effective user ID or the saved set-user-ID.

       Precisely the same holds for setegid() with "group" instead of
       "user".
```

Opening etc/shadow with setuid (1)

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void testFile() {
        char * filename = "/etc/shadow";
        FILE * f;
        f = fopen(filename, "r");
        if (f == NULL) {
                printf("failed!\n");
        }
        else {
                fclose(f);
                printf("OK\n");
        }
}
```

Opening etc/shadow with setuid (2)

```
int main(int argc, char ** argv) {
        int status;
        testFile();

        status = setuid(500);
        if (status < 0) {
                fprintf(stderr, "setuid failed!\n");
                return -1;
        }
        testFile();

        status = setuid(0);
        if (status < 0) {
                fprintf(stderr, "setuid failed!\n");
                return -1;
        }
        testFile();
}
```

- What happens when you execute this program as root ?

Now with seteuid

```c
int main(int argc, char ** argv) {
        int status;
        testFile();

        status = seteuid(500);
        if (status < 0) {
                fprintf(stderr, "setuid failed!\n");
                return -1;
        }
        testFile();

        status = seteuid(0);
        if (status < 0) {
                fprintf(stderr, "setuid failed!\n");
                return -1;
        }
        testFile();
}
```

- What happens when you execute this program as root ?

Safe usage of setuid

- Always check setuid return code
- Use seteuid to temporarily drop permissions
- Can drop additional permissions with setsid and setgroups
- Do not forget group permissions
- Close all file descriptors you do not need anymore
  - Permissions not checked at each read and write, only when file is opened

Outline

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

Remember : Kernel space vs user space

- Virtual memory divided into kernel space and user space
- Kernel space is for privileged operating system kernel, kernel extensions, and most device drivers
- User space is where applications and some drivers execute
- This protects memory and hardware from both malicious and buggy software
- **System calls** allow users to call kernel operations

System calls (syscalls)

- System calls allow users to call kernel operations
  - Interface between hardware and user
  - Somewhat restrict operations allowed
  - Hide hardware changes over time
- Examples are open, write, read, fstat, socket, bind, accept
- See `http://man7.org/linux/man-pages/man2/syscalls.2.html` for full list
- Wrapper functions provided by C library implementations
- Security risk !

seccomp

- seccomp = secure computing mode
- Goal : restrict the set of available system calls for process
- Two modes : basic/strict mode and advanced/filter mode

seccomp (2)

SECCOMP(2) Linux Programmer's Manual SECCOMP(2)

NAME top

seccomp - operate on Secure Computing state of the process

SYNOPSIS top

```
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>
#include <linux/signal.h>
#include <sys/ptrace.h>

int seccomp(unsigned int operation, unsigned int flags, void *args);
```

DESCRIPTION top

The **seccomp**() system call operates on the Secure Computing (seccomp) state of the calling process.

Currently, Linux supports the following *operation* values:

SECCOMP_SET_MODE_STRICT
The only system calls that the calling thread is permitted to make are read(2), write(2), _exit(2) (but not exit_group(2)), and sigreturn(2). Other system calls result in the delivery of a **SIGKILL** signal. Strict secure computing mode is useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket.

Basic seccomp (strict mode)

- Secure computing mode : program can only call
  - exit
  - sigreturn
  - read on already open files
  - write on already open files
- When attempting any other system call,
  kernel will terminate the process with SIGKILL
- One-way mode transition : process will never be able to make other system calls later
- Defense-in-depth : limit damages of potential attacks

Basic Seccomp : example

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>

int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp strict mode...\n");
        prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open file for reading...\n");
        int input = open("output.txt", O_RDONLY);
}
```

Code source : `https://gist.github.com/mstemm/3e29df625052616fffcd667ff59bf32a`

PRCTL : process control library

PRCTL(2) Linux Programmer's Manual PRCTL(2)

NAME top

prctl - operations on a process

SYNOPSIS top

```
#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3,
          unsigned long arg4, unsigned long arg5);
```

DESCRIPTION top

prctl() is called with a first argument describing what to do (with values defined in *<linux/prctl.h>*), and further arguments with a significance depending on the first one. The first argument can be:

PR_SET_SECCOMP (since Linux 2.6.23)

Set the secure computing (seccomp) mode for the calling thread, to limit the available system calls. The more recent seccomp(2) system call provides a superset of the functionality of **PR_SET_SECCOMP**.

The seccomp mode is selected via *arg2*. (The seccomp constants are defined in *<linux/seccomp.h>*.)

With *arg2* set to **SECCOMP_MODE_STRICT**, the only system calls that the thread is permitted to make are read(2), write(2), _exit(2) (but not exit_group(2)), and sigreturn(2). Other system calls result in the delivery of a **SIGKILL** signal. Strict secure computing mode is useful for number-crunching applications that may need to execute untrusted byte code, perhaps obtained by reading from a pipe or socket. This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

With *arg2* set to **SECCOMP_MODE_FILTER** (since Linux 3.5), the system calls allowed are defined by a pointer to a Berkeley Packet Filter passed in *arg3*. This argument is a pointer to *struct sock_fprog*; it can be designed to filter arbitrary system calls and system call arguments. This mode is available only if the kernel is configured with **CONFIG_SECCOMP_FILTER** enabled.

If **SECCOMP_MODE_FILTER** filters permit fork(2), then the seccomp mode is inherited by children created by fork(2); if execve(2) is permitted, then the seccomp mode is preserved across execve(2). If the filters permit **prctl**() calls, then additional filters can be added; they are run in order until the first non-allow result is seen.

For further information, see the kernel source file *Documentation/userspace-api/seccomp_filter.rst* (or *Documentation/prctl/seccomp_filter.txt* before Linux 4.13).

Basic Seccomp : example

- When executing previous code :

```
Calling prctl() to set seccomp strict mode...
Writing to an already open file...
Trying to open file for reading...
Killed
```

seccomp-bpf

- BPF = Berkeley packet filter
- seccomp extension using BPF policy syntax
  - Finer filtering of system calls
  - Filtering on parameters as well
    (such as writing only on some files)
  - Options to either kill the process, block illegal syscalls, or send warnings
- Convenient interface via libseccomp

libseccomp

LIBSECCOMP

https://github.com/seccomp/libseccomp

cii best practices passing | build passing | coverage 89%

The libseccomp library provides an easy to use, platform independent, interface to the Linux Kernel's syscall filtering mechanism. The libseccomp API is designed to abstract away the underlying BPF based syscall filter language and present a more conventional function-call based filtering interface that should be familiar to, and easily adopted by, application developers.

seccomp-bpf : example

```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>
#include "seccomp-bpf.h"

void install_syscall_filter()
{
        ...
}

int main(int argc, char **argv)
{
        ...
}
```

seccomp-bpf : example

```c
void install_syscall_filter()
{
        struct sock_filter filter[] = {
                /* Validate architecture. */
                VALIDATE_ARCHITECTURE,
                /* Grab the system call number. */
                EXAMINE_SYSCALL,
                /* List allowed syscalls. We add open() to the set of
                   allowed syscalls by the strict policy, but not
                   close(). */
                ALLOW_SYSCALL(rt_sigreturn),
#ifdef __NR_sigreturn
                ALLOW_SYSCALL(sigreturn),
#endif
                ALLOW_SYSCALL(exit_group),
                ALLOW_SYSCALL(exit),
                ALLOW_SYSCALL(read),
                ALLOW_SYSCALL(write),
                ALLOW_SYSCALL(open),
                KILL_PROCESS,
        };
        struct sock_fprog prog = {
                .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
                .filter = filter,
        };

        assert(prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == 0);

        assert(prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == 0);
}
```

Notes on PRCTL calls

```
SECCOMP_SET_MODE_FILTER
       The system calls allowed are defined by a pointer to a
       Berkeley Packet Filter (BPF) passed via args.  This argument
       is a pointer to a struct sock_fprog; it can be designed to
       filter arbitrary system calls and system call arguments.  If
       the filter is invalid, seccomp() fails, returning EINVAL in
       errno.

       If fork(2) or clone(2) is allowed by the filter, any child
       processes will be constrained to the same system call filters
       as the parent.  If execve(2) is allowed, the existing filters
       will be preserved across a call to execve(2).

       In order to use the SECCOMP_SET_MODE_FILTER operation, either
       the caller must have the CAP_SYS_ADMIN capability in its user
       namespace, or the thread must already have the no_new_privs
       bit set.  If that bit was not already set by an ancestor of
       this thread, the thread must make the following call:

           prctl(PR_SET_NO_NEW_PRIVS, 1);

       Otherwise, the SECCOMP_SET_MODE_FILTER operation will fail and
       return EACCES in errno.  This requirement ensures that an
       unprivileged process cannot apply a malicious filter and then
       invoke a set-user-ID or other privileged program using
       execve(2), thus potentially compromising that program.  (Such
       a malicious filter might, for example, cause an attempt to use
       setuid(2) to set the caller's user IDs to non-zero values to
       instead return 0 without actually making the system call.
       Thus, the program might be tricked into retaining superuser
       privileges in circumstances where it is possible to influence
       it to do dangerous things because it did not actually drop
       privileges.)

       If prctl(2) or seccomp() is allowed by the attached filter,
       further filters may be added.  This will increase evaluation
       time, but allows for further reduction of the attack surface
       during execution of a thread.

       The SECCOMP_SET_MODE_FILTER operation is available only if the
       kernel is configured with CONFIG_SECCOMP_FILTER enabled.
```

```
PR_SET_NO_NEW_PRIVS (since Linux 3.5)
       Set the calling thread's no_new_privs bit to the value in
       arg2.  With no_new_privs set to 1, execve(2) promises not to
       grant privileges to do anything that could not have been done
       without the execve(2) call (for example, rendering the set-
       user-ID and set-group-ID mode bits, and file capabilities non-
       functional).  Once set, this bit cannot be unset.  The setting
       of this bit is inherited by children created by fork(2) and
       clone(2), and preserved across execve(2).

       Since Linux 4.10, the value of a thread's no_new_privs bit can
       be viewed via the NoNewPrivs field in the /proc/[pid]/status
       file.

       For more information, see the kernel source file
       Documentation/userspace-api/no_new_privs.rst (or
       Documentation/prctl/no_new_privs.txt before Linux 4.13).  See
       also seccomp(2).
```

When executing a program, if the setuid bit is set on the program file pointed to by filename, then the effective user ID of the calling process is normally changed to that of the owner of the program file. The PR_SET_NO_NEW_PRIVS bit prevents that.

seccomp-bpf : example

```
int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp with filter...\n");
        install_syscall_filter();

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("Trying to close the file...\n");
        close(input);
}
```

Code source : gist.github.com/mstemm/1bc06c52abb7b6b4feef79d7bfff5815#file-seccomp_policy-c

seccomp-bpf : example

- When executing previous code :

```
Calling prctl() to set seccomp with filter...
Writing to an already open file...
Trying to open file for reading...
Trying to close the file...
Bad system call
```

seccomp applications : sandboxing

- Sandbox : mechanism for separating running programs, to mitigate system failures or software vulnerabilities from spreading (defense-in-depth)
- seccomp and seccomp-bpf applications :
  - Docker containers
  - OpenSSH
  - Used in Chrome to sandbox Adobe Flash Player
  - Firefox
  - Firejail : linux sandbox program
  - Tor
  - ...

Docker

- Open source project that enables software to run inside of isolated containers
- Docker containers use resource isolation and separate namespaces to isolate the application's view of the operating system

Outline

Unix Security Features

Setuid programs

Restricting system calls with seccomp

Summary

Summary

- Processes have real/effective/saved user and group IDs
- Access rights to files determined by Effective IDs
- Setuid/seteuid can give executer of a program the rights of its owner... must be used with care !
- System calls give you access to kernel operations in a restricted way ; can be restricted further using seccomp
- Other operating systems ? see group presentations !

References

- David Wheeler, Secure Programming HOWTO
- Dowd, McDonald, Schuh, The art of Software Security Asssessment, Chapters 9-10
- Matt Bishop, How To Write a Setuid Program, `nob.cs.ucdavis.edu/bishop/secprog/1987-sproglogin.pdf`
- Using simple seccomp filters, `outflux.net/teach-seccomp/`