Changelog

26 April 2021: chroot ls: add first missing chroot command; consistently use /tmp/example

26 April 2021: mount namespaces API: add mount commands that would be needed to actually run ls

26 April 2021: Linux namespaces (3): mention that a chroot will happen in addition to mounting things; be more explicit about how user namespaces change things, opting out of sudo, effectively

last time

Rust options for "smart pointer" types

other language enforcement ideas: e.g. constant-time

what programs are allowed to do vs. what programs need

- goal: "least privilege"

privilege separation as application design

- examples: Chrome, OpenSSH

mechanism: system call filtering

problem: more precise system call filtering?

challenge/final logistics

CHALLENGE assignment plan:

- 9% of final grade (= approx. 2 homeworks)
- 7 "make this program output X" problems
- solve any 5 for full credit
- expect to release no later than 30 April
- due 12 May 2021 @ 9pm

FINAL quiz

- 6% of final grade
- focused on material that can't be covered by CHALLENGE
  intending something that won't take longer than 90 minutes
  - (it varies how good I am meeting that target)
- released 12 May 2021 @ 9pm
- due 13 May 2021 @ 9pm
- (our official final period: 2-5pm 13 May)

challenge assistance policy

Please do not discuss or expect TAs (or me) to answer questions about what strategy you should apply to particular challenges. You are responsible for figuring this out yourself.

You may, however, ask TAs or share general information about how to identify whether an exploit technique is applicable to a particular program or about how to apply an exploit technique to other executables.

We will supply reference solutions to homework assignments.

Linux system call filtering: detailed

Linux supports more fine-grained system call filtering

using BPF (Berkeley Packet Filter) programming language

- compiled in the kernel to assembly to check system calls

can check system call argument values, but...

- problems with pointer arguments
- too many system calls

Linux system call: open

```
open("foo.txt", O_RDONLY);
```

parameters:

- system call number: 2 (“open”)
- argument 1: 0x7fffe983 (address of string “foo.txt”)
- argument 2: 0 (value of “O_RDONLY”)

very problematic to filter using BPF interface

can deal with using ‘ptrace’ — Linux debugging interface

- BPF can trigger something like a debugger breakpoint
- breakpoint wakes up monitor program (attached like debugger)
- ‘monitor’ program can perform system call on program’s behalf

filtering system calls?

example: video player VLC playing a local file on my laptop

uses 73 unique kinds of system calls

opens many files that are not the video file

- libraries
- fonts
- configuration files
- translations of messages

can I limit the files my video player can read?

how do I come up with a useful filter?

shared services?

often programs do operations by talking to "server" program

- example: GUI management on Linux (X11 or Wayland), OS X (WindowServer)
- example: mixing sound from multiple applications
- ...

whole extra set of calls to sanitize

- when to allow "get keyboard input" for GUI
- when to allow "get microphone input" for sound manager
- making sure one isn't manipulating wrong program's windows?

also, server programs might have security problems

- common "sandbox escape"

exercise: app confinement options

sandboxed applications want to access display server

which option seems best for security/performance?

A. proxy for protocol display server supports natively that filters display calls

B. custom protocol that sends bitmaps + receives inputs, plus copy of display server runs with application

C. divide application into UI and non-UI part, sandbox just the non-UI part

D. have application take over screen when running, give its own display server

SELinux

Security **E**nhanced Linux

"Mandatory Access Control" system for the Linux

> mandatory: can be configured to require enumeration of files programs can access
> (versus normally: specify what files programs can't access)

not necessairily run in mandatory control mode

programs run in particular "domain"

objects (files, port numbers, other programs, etc.) can be assigned labels

rules about what labels programs are allowed to access

viewing/assigning labels (1)

```
$ ls -Z /var/log/lastlog
-rw-r--r--. root root system_u:object_r:lastlog_t:s0
```

above: default Red Hat Linux/CentOS configuration

system user

object role

lastlog type

```
$ chcon --type=newtype_t some_file
```

assigning labels (2)

labels via: “file context mapping”

```
$ semanage fcontext --add --type web_files_t '/var/w
$ restorecon -R -v /var/www/html
```

pattern matching rules set *default* labels

restorecon — switch to default labels, applying rules

assigning rules

subset of default rules for Apache httpd (webserver):

```
define(`read_files_pattern',`
  allow $1 $2:dir search_dir_perms;
  allow $1 $3:file read_file_perms;
')
...
define(`read_lnk_files_pattern',`
  allow $1 $2:dir search_dir_perms;
  allow $1 $3:lnk_file read_lnk_file_perms;
')
...
allow httpd_t httpd_config_t:dir list_dir_perms;
read_files_pattern(httpd_t, httpd_config_t, httpd_config_t)
read_lnk_files_pattern(httpd_t, httpd_config_t, httpd_config_t
```

httpd_t: ‘type’ for webserver process

changing what programs can name

seccomp, SELinux: program tries to access X, checks if allowed

alternate idea: changing what Xs program can name

Unix filesystems and mounting

my Linux desktop has two disks:

- `/` — an SSD
- `/mnt/extradisk` — a hard drive

hard drive appears as *subdirectory* of SSD

subdirectory called a *mount point*

per-process root

on Unix: each process tracks its own root directory (/)

can be changed with chroot() system call

- command-line tool to access: `chroot`

usage: can isolate program from other files on system

- example: limit what public file server can access?

chroot ls

```
# mkdir /tmp/example
# cp /bin/ls /tmp/example/ls
# chroot /tmp/example /ls
chroot: failed to run command ‘/ls’: No such file or directory
# cp -r /lib64 /tmp/example/lib64
# mkdir -p /tmp/example/lib
# cp -r /lib/x86_64-linux-gnu /tmp/example/lib/x86_64-linux-gnu
# chroot /tmp/example /ls
/ls: error while loading shared libraries: libpcre2-8.so.0: cannot o
# cp /usr/lib/x86_64-linux-gnu/libpcre2-8* /tmp/example/lib/x86_64-l
# chroot /tmp/example /ls /
lib  lib64  ls
# chroot /tmp/example /ls /..
lib  lib64  ls
#
```

chroot escapes

chroot prevents accessing files outside the new /

but root (system adminstrator) user in chroot can access disks, etc.

typical usage: combine chroot with extra user

chroot impracticality

some things make chroot impractical in general:

seems like one needs extra copies of most of the system

hard to communicate between separate roots

requires administrator permissions to configure

- dangerous to let normal users configure b/c they could confuse priviliged (set-user-ID) programs like `sudo`

exercise

what scenarios does chroot make most/least sense for?

A. the rendering part of web browser

B. a web server

C. a media player

D. a network time server (for other machines to set their clocks)

Linux namespaces (1)

Linux: alternate sandboxing features

"namespaces" for other resources

chroot: each process has own idea of root directory

- change to OS: look up root directory in process, not global variable

can apply this to other resources:

- what filesystems (disks) are available
- what network devices are available
- what user identifier numbers are
- ...

Linux namespaces (2)

user namespace:

can run programs with new view of users:

inside namespace: running as root

outside namespace: root translated to innocent user ID

allows running programs that expect different users

...without changes, but without giving special permissions

mechanism: reassign user ID numbers in kernel

aside: Linux clone(), unshare() syscalls

Linux clone system call: start new process (or thread)

flags to specify environment of new process

these flags can include "make a new namespace of a type"

```
int id = clone(start_function, ..., CLONE_NEWUSER | other_flags);
```

above option: new user namespace for new process

alternative: for changing current process's namespace:

```
unshare(CLONE_NEWUSER);
```

user namespaces API

Linux: users identified by numerical *user IDs* (UIDs)

with user namespaces:

control file `/proc/PROCESS-ID/UID_MAP` contains lines like:

- `0 1000 2` — UID 0–1 maps to UID 1000–1001
- `1000 2000 100` — UID 1000-1100 maps to UID 2000–2100

can write to that file to reconfigure (if enough permissions)

Linux namesapces (3)

mount namespaces:

Unix: mounting disk = making the contents of the disk available as directories+files

different idea of what filesystems are available

can be setup with *bind mounts* to "real FS"

but otherwise: no access to directories outside mount namespace

normally requires root — but special case with user namespaces

mount namespaces API

from command line:

```
    # runs shell (/bin/sh) in new mount namesapce
shell1$ unshare --mount /bin/sh

    # setup directories in /tmp/workdir and make them aliases of th
    # these aliases will only exist for processes in mount namespace
shell2$ mkdir -p /tmp/workdir/bin
shell2$ mkdir -p /tmp/workdir/lib
shell2$ mkdir -p /tmp/workdir/usr
shell2$ mkdir -p /tmp/workdir/current
shell2$ mount -o bind,ro /bin /tmp/workdir/bin
shell2$ mount -o bind,ro /lib /tmp/workdir/lib
shell2$ mount -o bind,ro /usr /tmp/workdir/usr
shell2$ mount -o bind /home/someuser /tmp/workdir/current

    # start new shell with the root directory being /tmp/workdir
shell2$ chroot /tmp/workdir /bin/sh
shell3$ cd /
shell3$ /bin/ls
bin     current     lib     usr
```

Linux namespaces (3)

user namespace and mount namespace together:

run program in new user namespace

map regular root (in namespace) to regular user

- "opts out" of programs like sudo

move to new mount namespace

setup bind mounts + chroot

- special case: allowed because root in user namespce
- can't get "real" root (administrator) privileges ever

run program with subset of available files

Linux namespaces (4)

other resources with namespaces

- network — common usage: virtual network device for set processes
- hostname (“UTS”)
- process identifiers
- control groups (resource limits for memory, CPU usage, disk I/O, etc.)

Linux sandboxing programs, generally

docker, lxc, lxd, containerd

- use namespaces to create "container" with own copy of OS libraries, services
- but containers share OS 'kernel' and potentially files with host unlike VM
- (might also have options to use other ways of getting this functionality — VM's, etc.)

bubblewrap, firejail

- use Linux namespace tools + "bind mounts" to give programs only subset of files, etc.
- firejail has option of running a "proxy" windowing system server

SELinux's sandbox

- uses Security Enhanced Linux's mandatory access controls instead of Linux namespaces
- includes option for "proxy" for windoing system server

containers

Linux's seccomp + namespaces + SELinux commonly used to implement containers

(plus cgroups (control groups) for performance isolation)

usual goal: looks like virtual machine, but much lower overhead

examples: Docker, Kubernetes

(note: these may also support other ways of creating 'lightweight VMs')

runc bug

2019 bug in Docker, other container implementations (CVE-2019-5736)

- blog post for vulnerability finders:
  `https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html`

bug setup:

- user starts malicious container X
- user tells docker to start a new command in malicious container X
- malicious container X hijacks the "new command" starting program
- hijacked program used to access stuff outside container

part of problem: Docker and others weren't using user namespaces at the time

- compatability problems

runc bug

2019 bug in Docker, other container implementations (CVE-2019-5736)

- blog post for vulnerability finders:
  `https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html`

bug setup:

- user starts malicious container X
- user tells docker to start a new command in malicious container X
- malicious container X hijacks the "new command" starting program
- hijacked program used to access stuff outside container

part of problem: Docker and others weren't using user namespaces at the time

- compatability problems

setup: /proc/PID

Linux provides /proc directory to access info about programs

used for implementing process list utils, debugging

- needed to make a functional container

subdirectory for each process in current container

- process ID PID has /proc/PID subdirectory
- /proc/self is alias for current process's subdirectory

included is /proc/PID/exe file — alias for executable file

running a command in existing container

to run command X in existing container:

step 1: switch current process to that container

step 2: execute command X

running a command in existing container

to run command X in existing container:

step 1: switch current process to that container

code in container can access /proc here?

including overwriting /proc/self/exe!

- which is a program run as root!

step 2: execute command X

partial fix

can disable access to /proc/PID/exe (and related things)

system call: `prctl(PR_SET_DUMPABLE, 0)`

but...the run-in-container tool did this for a while

partial fix

can disable access to /proc/PID/exe (and related things)

system call: `prctl(PR_SET_DUMPABLE, 0)`

but...the run-in-container tool did this for a while

problem: this gets reset on executing a new program

and attacker could make the new program be /proc/PID/exe

- one mechanism: symbolic links (file aliases)

but change dynamic linking setup to run attacker code

...which accesses /proc/self/exe

full fix

make single-use copy of start-in-container tool each time command run

- in-memory file

...so modifying it doesn't change anything

- (but it's also protected from modification)

other solutions:

- make executable non-writable (e.g. SELinux, don't run container as root)