Introduction to Linux: **Advanced Permissions**

Philipp Fruck & Pius Walter

Advanced Permissions

Goal of this lecture

- Learn how to safely elevate privileges
- Understand fine-grained access control beyond classic permissions (ACLs)
- Understand Linux capabilities as a safer alternative to full root access

Why?

- Classic permissions are often not granular enough
- Minimizing granted privileges reduces the security impact of bugs and exploits

sudo

Security

- `sudo` is typically a setuid root binary
- Runs with effective UID 0, regardless of the invoking user's privileges
- Part of the system's trusted computing base
- Vulnerabilities can lead to local privilege escalation
- Overly permissive rules can effectively grant unrestricted root access

sudo

Security

- `sudo` is typically a setuid root binary
- Runs with effective UID 0, regardless of the invoking user's privileges
- Part of the system's trusted computing base
- Vulnerabilities can lead to local privilege escalation
- Overly permissive rules can effectively grant unrestricted root access

Configuration

- `/etc/sudoers` and `/etc/sudoers.d/`
- Defines who can execute what, as which user
- `visudo`: syntax checking and file locking
  - Use `sudo visudo` when editing!

```
# who  where=(as_whom:as_group)  options: cmd
# Allow wheel group to execute any command
%wheel  ALL=(ALL:ALL) ALL
# Allow command execution without password
%wheel  ALL=(ALL:ALL) NOPASSWD: ALL
```

sudo

Security

- `sudo` is typically a setuid root binary
- Runs with effective UID 0, regardless of the invoking user's privileges
- Part of the system's trusted computing base
- Vulnerabilities can lead to local privilege escalation
- Overly permissive rules can effectively grant unrestricted root access

Configuration

- `/etc/sudoers` and `/etc/sudoers.d/`
- Defines who can execute what, as which user
- `visudo`: syntax checking and file locking
  - Use `sudo visudo` when editing!

```
# who  where=(as_whom:as_group)  options: cmd
# Allow wheel group to execute any command
%wheel  ALL=(ALL:ALL) ALL
# Allow command execution without password
%wheel  ALL=(ALL:ALL) NOPASSWD: ALL
```

`sudo-rs`: Rust implementation of sudo

- Memory safety reduces certain risks, but does not eliminate privilege-escalation risk
- Shipped by Ubuntu by default as of 25.10 / 26.04 LTS
- Not 1:1 compatible with original `sudo`!

sudo

Security

- `sudo` is typically a setuid root binary
- Runs with effective UID 0, regardless of the invoking user's privileges
- Part of the system's trusted computing base
- Vulnerabilities can lead to local privilege escalation
- Overly permissive rules can effectively grant unrestricted root access

Configuration

- `/etc/sudoers` and `/etc/sudoers.d/`
- Defines who can execute what, as which user
- `visudo`: syntax checking and file locking
  - Use `sudo visudo` when editing!

```
# who  where=(as_whom:as_group)  options: cmd
# Allow wheel group to execute any command
%wheel  ALL=(ALL:ALL) ALL
# Allow command execution without password
%wheel  ALL=(ALL:ALL) NOPASSWD: ALL
```

`sudo-rs`: Rust implementation of sudo

- Memory safety reduces certain risks, but does not eliminate privilege-escalation risk
- Shipped by Ubuntu by default as of 25.10 / 26.04 LTS
- Not 1:1 compatible with original `sudo`!

Q: Which permission does this rule grant?
`ALL=(ALL:ALL) /usr/bin/vim /etc/groups`

sudo

Security

- `sudo` is typically a setuid root binary
- Runs with effective UID 0, regardless of the invoking user's privileges
- Part of the system's trusted computing base
- Vulnerabilities can lead to local privilege escalation
- Overly permissive rules can effectively grant unrestricted root access

Configuration

- `/etc/sudoers` and `/etc/sudoers.d/`
- Defines who can execute what, as which user
- `visudo`: syntax checking and file locking
  - Use `sudo visudo` when editing!

```
# who  where=(as_whom:as_group)  options: cmd
# Allow wheel group to execute any command
%wheel  ALL=(ALL:ALL) ALL
# Allow command execution without password
%wheel  ALL=(ALL:ALL) NOPASSWD: ALL
```

`sudo-rs`: Rust implementation of sudo

- Memory safety reduces certain risks, but does not eliminate privilege-escalation risk
- Shipped by Ubuntu by default as of 25.10 / 26.04 LTS
- Not 1:1 compatible with original `sudo`!

Q: Which permission does this rule grant?
`ALL=(ALL:ALL) /usr/bin/vim /etc/groups`

- Read and write the file as root
- But also: shell via `:sh`
  - Privilege escalation

run0

`run0` is the systemd (>=256) native privilege escalation tool. It uses `polkit` instead of `setuid`.

run0

`run0` is the systemd (>=256) native privilege escalation tool. It uses `polkit` instead of `setuid`.

Basic usage

Similar to `sudo`, but you might need `--via-shell`

```
run0 id
run0 -u daemon id
run0 printenv PATH
run0 --via-shell printenv PATH
```

run0

`run0` is the systemd (>=256) native privilege escalation tool. It uses `polkit` instead of `setuid`.

Basic usage

Similar to `sudo`, but you might need `--via-shell`

```
run0 id
run0 -u daemon id
run0 printenv PATH
run0 --via-shell printenv PATH
```

Sandboxing

```
# read-only except /tmp
run0 --property=ProtectSystem=strict \
    --property=ReadWritePaths=/tmp \
    touch /tmp/1
# Hide /home
run0 --property=ProtectHome=true ls /home
cd /tmp && !!
```

run0

`run0` is the systemd (>=256) native privilege escalation tool. It uses `polkit` instead of `setuid`.

Basic usage

Similar to `sudo`, but you might need `--via-shell`

```
run0 id
run0 -u daemon id
run0 printenv PATH
run0 --via-shell printenv PATH
```

Sandboxing

```
# read-only except /tmp
run0 --property=ProtectSystem=strict \
    --property=ReadWritePaths=/tmp \
    touch /tmp/1
# Hide /home
run0 --property=ProtectHome=true ls /home
cd /tmp && !!
```

> run0 does not automatically make commands safe. The actual isolation depends on the systemd properties applied.

Access Control Lists (ACLs)

Problem: Traditional permissions only allow control for a single **owner**, **group** and **others**

Solution: Access Control Lists (ACLs) let you define per-user or per-group permissions

Access Control Lists (ACLs)

Problem: Traditional permissions only allow control for a single **owner**, **group** and **others**

Solution: Access Control Lists (ACLs) let you define per-user or per-group permissions

```
# Show ACLs
$ getfacl secret.txt
# Add a new user permission
$ setfacl -m u:bob:r secret.txt
# Remove an ACL entry
$ setfacl -x u:bob secret.txt
# Notice the little + sign in ls -l
$ ls -l secret.txt
-rw-r--r--+ 1 demo demo 0 Nov 10 16:19 secret.txt
```

Access Control Lists (ACLs)

Problem: Traditional permissions only allow control for a single **owner**, **group** and **others**

Solution: Access Control Lists (ACLs) let you define per-user or per-group permissions

```
# Show ACLs
$ getfacl secret.txt
# Add a new user permission
$ setfacl -m u:bob:r secret.txt
# Remove an ACL entry
$ setfacl -x u:bob secret.txt
# Notice the little + sign in ls -l
$ ls -l secret.txt
-rw-r--r--+ 1 demo demo 0 Nov 10 16:19 secret.txt
```

```
$ getfacl secret.txt
# file: secret.txt
# owner: alice
# group: staff
user::rw-
user:bob:r--
group::r--
mask::r--
other::---
```

Access Control Lists (ACLs)

Problem: Traditional permissions only allow control for a single **owner**, **group** and **others**

Solution: Access Control Lists (ACLs) let you define per-user or per-group permissions

```
# Show ACLs
$ getfacl secret.txt
# Add a new user permission
$ setfacl -m u:bob:r secret.txt
# Remove an ACL entry
$ setfacl -x u:bob secret.txt
# Notice the little + sign in ls -l
$ ls -l secret.txt
-rw-r--r--+ 1 demo demo 0 Nov 10 16:19 secret.txt
```

```
$ getfacl secret.txt
# file: secret.txt
# owner: alice
# group: staff
user::rw-
user:bob:r--
group::r--
mask::r--
other::---
```

Default ACLs: If set on directories, ACLs apply to all new files inside: `setfacl -d -m u:bob:rw project/`

ACLs offer fine-grained control -> especially useful for multi-user servers.

Still, they are rather hidden and rarely used.

Linux Capabilities

Problem: `setuid root` gives full root privileges -> too powerful

Solution: **Linux Capabilities** split root privileges into fine-grained pieces

- Each process can get **only** the privileges it needs

Linux Capabilities

Problem: `setuid root` gives full root privileges -> too powerful

Solution: **Linux Capabilities** split root privileges into fine-grained pieces

- Each process can get **only** the privileges it needs

Example Capability	What it Allows	Example command
`CAP_NET_ADMIN`	Manage network interfaces, routing tables	`ip link set eth0 down`
`CAP_NET_RAW`	Use raw and packet sockets	`ping 1.1`
`CAP_SYS_ADMIN`	"Catch-all" - system-wide administration	`mount /dev/sda /mnt`
`CAP_DAC_OVERRIDE`	Bypass file read/write permission checks	`cat /root/secret.txt`
`CAP_SYS_TIME`	Change the system clock	`date -s "2026-11-10 12:00:00"`
`CAP_NET_BIND_SERVICE`	Bind to ports < 1024	`python3 -m http.server 80`

Linux Capabilities

Problem: `setuid root` gives full root privileges -> too powerful

Solution: **Linux Capabilities** split root privileges into fine-grained pieces

- Each process can get **only** the privileges it needs

Example Capability	What it Allows	Example command
`CAP_NET_ADMIN`	Manage network interfaces, routing tables	`ip link set eth0 down`
`CAP_NET_RAW`	Use raw and packet sockets	`ping 1.1`
`CAP_SYS_ADMIN`	"Catch-all" - system-wide administration	`mount /dev/sda /mnt`
`CAP_DAC_OVERRIDE`	Bypass file read/write permission checks	`cat /root/secret.txt`
`CAP_SYS_TIME`	Change the system clock	`date -s "2026-11-10 12:00:00"`
`CAP_NET_BIND_SERVICE`	Bind to ports < 1024	`python3 -m http.server 80`

> As always: See man capabilities for details

Overview: The Five Capability Sets in Linux

Permitted (P)

This is the list of all privileges a process is allowed to have. Anything not on this list can never be obtained by the process.

Overview: The Five Capability Sets in Linux

Permitted (P)

This is the list of all privileges a process is allowed to have. Anything not on this list can never be obtained by the process.

Effective (E)

The privileges that are currently active. Only these are checked by the kernel when, e.g., a program tries to access port 80 or modify system files.

Overview: The Five Capability Sets in Linux

Permitted (P)

This is the list of all privileges a process is allowed to have. Anything not on this list can never be obtained by the process.

Effective (E)

The privileges that are currently active. Only these are checked by the kernel when, e.g., a program tries to access port 80 or modify system files.

Inheritable (I)

The privileges that can be passed on to a child process (after `exec()`). This allows a program to give another program certain privileges.

Overview: The Five Capability Sets in Linux

Permitted (P)

This is the list of all privileges a process is allowed to have. Anything not on this list can never be obtained by the process.

Effective (E)

The privileges that are currently active. Only these are checked by the kernel when, e.g., a program tries to access port 80 or modify system files.

Inheritable (I)

The privileges that can be passed on to a child process (after `exec()`). This allows a program to give another program certain privileges.

Bounding (B)

This is the maximum limit of all possible privileges. Once a privilege is removed, it is permanently gone, even root cannot bring it back.

Overview: The Five Capability Sets in Linux

Permitted (P)

This is the list of all privileges a process is allowed to have. Anything not on this list can never be obtained by the process.

Effective (E)

The privileges that are currently active. Only these are checked by the kernel when, e.g., a program tries to access port 80 or modify system files.

Inheritable (I)

The privileges that can be passed on to a child process (after `exec()`). This allows a program to give another program certain privileges.

Bounding (B)

This is the maximum limit of all possible privileges. Once a privilege is removed, it is permanently gone, even root cannot bring it back.

Ambient (A)

Newer addition (since kernel 4.3): privileges that are automatically retained when starting a new program, if the program is not `setuid` and the privilege is present in both `P` and `I`. Used, for example, by containers or systemd.

Linux Capabilities

View capabilities

```
getcap /usr/bin/arping
```

Set or remove capabilities

```
# add effective + permitted net raw capability
run0 setcap cap_net_raw+ep /usr/bin/arping
# remove net raw again
run0 setcap cap_net_raw-ep /usr/bin/arping
# remove all capabilities
run0 setcap -r /usr/bin/arping
```

Linux Capabilities

View capabilities

```
getcap /usr/bin/arping
```

Set or remove capabilities

```
# add effective + permitted net raw capability
run0 setcap cap_net_raw+ep /usr/bin/arping
# remove net raw again
run0 setcap cap_net_raw-ep /usr/bin/arping
# remove all capabilities
run0 setcap -r /usr/bin/arping
```

Interactively

`capsh` can interactively set/drop permissions

```
run0 capsh --drop=cap_net_bind_service -- \
    -c 'python3 -m http.server 80'
```

Linux Capabilities

View capabilities

```
getcap /usr/bin/arping
```

Set or remove capabilities

```
# add effective + permitted net raw capability
run0 setcap cap_net_raw+ep /usr/bin/arping
# remove net raw again
run0 setcap cap_net_raw-ep /usr/bin/arping
# remove all capabilities
run0 setcap -r /usr/bin/arping
```

Interactively

`capsh` can interactively set/drop permissions

```
run0 capsh --drop=cap_net_bind_service -- \
    -c 'python3 -m http.server 80'
```

Revocation limitation

- Once a process has a capability, the kernel cannot revoke it mid-execution
- You must terminate or restart the process to remove privileges
  - Fun fact: This is different on BSD

Linux Capabilities

View capabilities

```
getcap /usr/bin/arping
```

Set or remove capabilities

```
# add effective + permitted net raw capability
run0 setcap cap_net_raw+ep /usr/bin/arping
# remove net raw again
run0 setcap cap_net_raw-ep /usr/bin/arping
# remove all capabilities
run0 setcap -r /usr/bin/arping
```

Interactively

`capsh` can interactively set/drop permissions

```
run0 capsh --drop=cap_net_bind_service -- \
    -c 'python3 -m http.server 80'
```

Revocation limitation

- Once a process has a capability, the kernel cannot revoke it mid-execution
- You must terminate or restart the process to remove privileges
  - Fun fact: This is different on BSD

More about ACLs and Linux Capabilities: https://media.ccc.de/v/why2025-192-linux-permissions-and-hardening

Exercise

Capabilities

Let's use `/usr/bin/tar` as an example:

1. Check if it currently has any capabilities
2. Give it a test capability (`CAP_DAC_READ_SEARCH`)

```
sudo setcap cap_dac_read_search+ep /usr/bin/tar
getcap /usr/bin/tar
```

3. Try reading a file you normally couldn't (as non-root)

```
tar -cvf /tmp/test.tar /tmp/rootfile.txt
```

4. **Important**: Remove the capability again with `sudo setcap -r /usr/bin/tar`
5. What could go wrong if tools like `tar` permanently kept this capability?

Access Control Lists

1. Create a file and set restrictive permissions

```
touch project.txt
chmod 600 project.txt
```

2. Add read access for another user (replace `<user>`)

```
sudo setfacl -m u:<user>:r project.txt
getfacl project.txt
```

3. Verify that `<user>` can now read the file even though group/others cannot.
4. Remove the ACL entry again

```
sudo setfacl -x u:<user> project.txt
```

Thank you!

Thank you for your attention!

Don't forget the feedback in Moodle please!