# OS Security

## Authorization

Radboud University Nijmegen, The Netherlands

Winter 2015/2016

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him
- Alternative: operation authentication, only feasible for very few, important operations

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him
- Alternative: operation authentication, only feasible for very few, important operations
- Three approaches to authentication:
  - By "what you know" (typically a password)
  - By "what you have" (typically a key, token, or smart-card)
  - By "what you are" (biometrics, e.g, fingerprint, iris scan)

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him
- Alternative: operation authentication, only feasible for very few, important operations
- Three approaches to authentication:
  - By "what you know" (typically a password)
  - By "what you have" (typically a key, token, or smart-card)
  - By "what you are" (biometrics, e.g, fingerprint, iris scan)
- Classical UNIX/Linux authentication through user data in `/etc/passwd` and `/etc/shadow`

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him
- Alternative: operation authentication, only feasible for very few, important operations
- Three approaches to authentication:
  - By "what you know" (typically a password)
  - By "what you have" (typically a key, token, or smart-card)
  - By "what you are" (biometrics, e.g, fingerprint, iris scan)
- Classical UNIX/Linux authentication through user data in `/etc/passwd` and `/etc/shadow`
- Flexible mechanism for managing authentication: PAM
  - Authentication modules in `/lib/security/`
  - Per-application configuration files in `/etc/pam.d/`
  - Library `libpam` as easy mechanism for applications to use PAM

# A short recap

- Authentication establishes a mapping between entities (users) and intended operations
- Typical approach: user authentication:
  - User logs into the system
  - Processes started by the user are linked to him
- Alternative: operation authentication, only feasible for very few, important operations
- Three approaches to authentication:
  - By “what you know” (typically a password)
  - By “what you have” (typically a key, token, or smart-card)
  - By “what you are” (biometrics, e.g, fingerprint, iris scan)
- Classical UNIX/Linux authentication through user data in `/etc/passwd` and `/etc/shadow`
- Flexible mechanism for managing authentication: PAM
  - Authentication modules in `/lib/security/`
  - Per-application configuration files in `/etc/pam.d/`
  - Library `libpam` as easy mechanism for applications to use PAM
- Authentication even more tricky in networked environments
- State of the art: LDAP and Kerberos

# Protection rings

- OS needs to control access to resources
- Idea: Access to resources only for highly-priveleged code
- Non-priveleged code needs to ask the OS to perform operations on resources

# Protection rings

- OS needs to control access to resources
- Idea: Access to resources only for highly-priveleged code
- Non-priveleged code needs to ask the OS to perform operations on resources
- Separate code in *protection rings*
- Ring 0: OS *kernel*
- Outer rings: less privileged software (drivers, userspace programs)

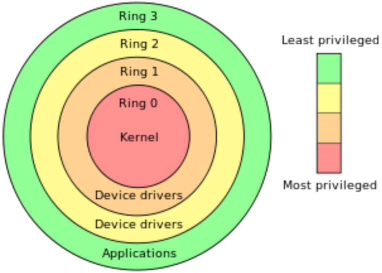



Image source: http://en.wikipedia.org/wiki/Protection_ring

# Protection rings in Linux

- Protection rings are supported by hardware
- Certain instructions can only be executed by privileged (ring-0) code
- X86 and AMD64 support 4 different rings (ring 0–3)
- Trying to execute a ring-0 instruction from ring-3 results in SIGILL (illegal instruction)
- Idea:
  - OS kernel (memory and process management) run in ring 0
  - Device drivers run in ring 1 and 2
  - Userspace software runs in ring 3

# Protection rings in Linux

- Protection rings are supported by hardware
- Certain instructions can only be executed by privileged (ring-0) code
- X86 and AMD64 support 4 different rings (ring 0–3)
- Trying to execute a ring-0 instruction from ring-3 results in SIGILL (illegal instruction)
- Idea:
  - OS kernel (memory and process management) run in ring 0
  - Device drivers run in ring 1 and 2
  - Userspace software runs in ring 3
- Linux (and Windows) use a simpler *supervisor-mode* model:
  - Operating system runs with supervisor flag enabled (ring 0)
  - Userspace programs run with supervisor flag disabled (ring 3)

# Protection rings in Linux

- Protection rings are supported by hardware
- Certain instructions can only be executed by privileged (ring-0) code
- X86 and AMD64 support 4 different rings (ring 0–3)
- Trying to execute a ring-0 instruction from ring-3 results in SIGILL (illegal instruction)
- Idea:
  - OS kernel (memory and process management) run in ring 0
  - Device drivers run in ring 1 and 2
  - Userspace software runs in ring 3
- Linux (and Windows) use a simpler *supervisor-mode* model:
  - Operating system runs with supervisor flag enabled (ring 0)
  - Userspace programs run with supervisor flag disabled (ring 3)
  - Call ring-0 code *kernel space*
  - Call ring-3 code *user space*

# System calls and `strace`

- Transition from user space to kernel space through well-defined interface
- Interface is a set of *system calls* (syscalls)
- A system call is a request from user space to the OS to perform a certain operation

# System calls and `strace`

- Transition from user space to kernel space through well-defined interface
- Interface is a set of *system calls* (syscalls)
- A system call is a request from user space to the OS to perform a certain operation
- Access to system calls is typically implemented through the standard library
- Examples:
  - `write` function defined in `unistd.h` is wrapper around `write` syscall
  - `execve` function defined in `unistd.h` is wrapper around `execve` syscall

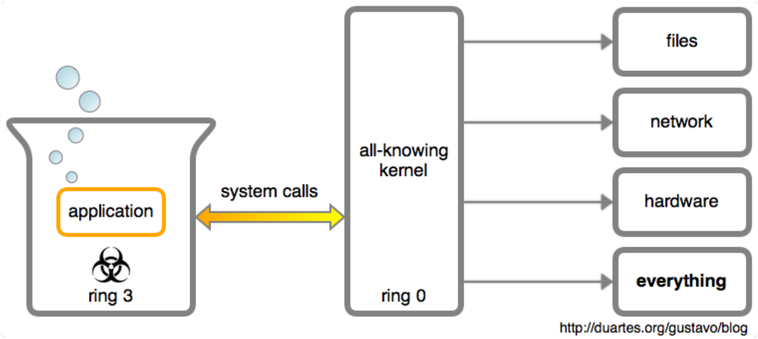
# System calls and `strace`

- Transition from user space to kernel space through well-defined interface
- Interface is a set of *system calls* (syscalls)
- A system call is a request from user space to the OS to perform a certain operation
- Access to system calls is typically implemented through the standard library
- Examples:
  - `write` function defined in `unistd.h` is wrapper around `write` syscall
  - `execve` function defined in `unistd.h` is wrapper around `execve` syscall
- Sometimes don't use system calls that directly, e.g., `printf` also calls `write`

# System calls and `strace`

- Transition from user space to kernel space through well-defined interface
- Interface is a set of *system calls* (syscalls)
- A system call is a request from user space to the OS to perform a certain operation
- Access to system calls is typically implemented through the standard library
- Examples:
  - `write` function defined in `unistd.h` is wrapper around `write` syscall
  - `execve` function defined in `unistd.h` is wrapper around `execve` syscall
- Sometimes don't use system calls that directly, e.g., `printf` also calls `write`
- Can print (trace) all syscalls of a program: `strace`
- Very helpful for understanding what's happening "behind the scenes"

# Applications and the OS



# Kernel modules

- Processes belonging to `root` can do anything

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**
- `root` processes do not run in kernel space
- `root` processes need syscalls to access resources

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**
- `root` processes do not run in kernel space
- `root` processes need syscalls to access resources
- What if there is no syscall for a certain operation?

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**
- `root` processes do not run in kernel space
- `root` processes need syscalls to access resources
- What if there is no syscall for a certain operation?
- Example: enable userspace access to hardware cycle counter on ARM processors

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**
- `root` processes do not run in kernel space
- `root` processes need syscalls to access resources
- What if there is no syscall for a certain operation?
- Example: enable userspace access to hardware cycle counter on ARM processors
- Answer: Modify OS kernel (add syscall), reboot

# Kernel modules

- Processes belonging to `root` can do anything **in userspace**
- `root` processes do not run in kernel space
- `root` processes need syscalls to access resources
- What if there is no syscall for a certain operation?
- Example: enable userspace access to hardware cycle counter on ARM processors
- Answer: Modify OS kernel (add syscall), reboot
- Better answer: Modify OS kernel *at runtime*
- Linux kernel typically allows to load *kernel modules*
- Modules run in kernel space (ring 0)
- Load module into kernel with program `insmod`

# A kernel module example

```
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");

#define DEVICE_NAME "enableccnt"

static int enableccnt_init(void)
{
  printk(KERN_INFO DEVICE_NAME " starting\n");
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(1));
  return 0;
}

static void enableccnt_exit(void)
{
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(0));
  printk(KERN_INFO DEVICE_NAME " stopping\n");
}

module_init(enableccnt_init);
module_exit(enableccnt_exit);
```

# Files

- Persistent data on background storage is organized in *files*
- Files are logical units of information organized by a *file system*
- Files have names and additional associated information:
  - Date and time of last access
  - Date and time of last modification
  - Access-permission-related information

# Files

- Persistent data on background storage is organized in *files*
- Files are logical units of information organized by a *file system*
- Files have names and additional associated information:
  - Date and time of last access
  - Date and time of last modification
  - Access-permission-related information
- Files are logically organized in a tree hierarchy of *directories*
- The file system maps logical information to bits and bytes on the storage device
- The file system runs in kernel space (typically through device drivers)
- Access to files goes through system calls

# "Everything is a file"

- Design principle of UNIX (and Linux): every persistent resource is accessed through a file handle
- A file handle is an integer, which is mapped to a resource
- Mapping is established per process in a kernel-managed file-descriptor table

# "Everything is a file"

- Design principle of UNIX (and Linux): every persistent resource is accessed through a file handle
- A file handle is an integer, which is mapped to a resource
- Mapping is established per process in a kernel-managed file-descriptor table
- Special file handles for (almost) every process:

| **Integer value** | **Name/Meaning** | **<stdio.h> file stream** |
|---:|---:|---|
| 0 | Standard input | stdin |
| 1 | Standard output | stdout |
| 2 | Standard error | stderr |

# “Everything is a file”

- Design principle of UNIX (and Linux): every persistent resource is accessed through a file handle
- A file handle is an integer, which is mapped to a resource
- Mapping is established per process in a kernel-managed file-descriptor table
- Special file handles for (almost) every process:

| **Integer value** | **Name/Meaning** | **<stdio.h> file stream** |
|---:|---:|---|
| 0 | Standard input | stdin |
| 1 | Standard output | stdout |
| 2 | Standard error | stderr |

- Consequence of “everything is a file”:
  - User-space processes can operate on files *only* through syscalls
  - OS can check for each syscall (kernel-space operation), whether the operation is permitted

# "Everything is a file"

- Design principle of UNIX (and Linux): every persistent resource is accessed through a file handle
- A file handle is an integer, which is mapped to a resource
- Mapping is established per process in a kernel-managed file-descriptor table
- Special file handles for (almost) every process:

| **Integer value** | **Name/Meaning** | **<stdio.h> file stream** |
|---:|---:|---|
| 0 | Standard input | stdin |
| 1 | Standard output | stdout |
| 2 | Standard error | stderr |

- Consequence of "everything is a file":
  - User-space processes can operate on files *only* through syscalls
  - OS can check for each syscall (kernel-space operation), whether the operation is permitted
  - (User-space programs also operate on memory, but that's for next lecture)

# File-related syscalls

- `open()`: Open a file and return a file handle

# File-related syscalls

- `open()`: Open a file and return a file handle
- `read()`: Read a number of bytes from a file handle into a buffer

# File-related syscalls

- `open()`: Open a file and return a file handle
- `read()`: Read a number of bytes from a file handle into a buffer
- `write()`: Write a number of bytes from a buffer to the file handle

# File-related syscalls

- `open()`: Open a file and return a file handle
- `read()`: Read a number of bytes from a file handle into a buffer
- `write()`: Write a number of bytes from a buffer to the file handle
- `close()`: Close the file handle

# File-related syscalls

- `open()`: Open a file and return a file handle
- `read()`: Read a number of bytes from a file handle into a buffer
- `write()`: Write a number of bytes from a buffer to the file handle
- `close()`: Close the file handle
- `lseek()`: Change position in the file handle

# File-related syscalls

- `open()`: Open a file and return a file handle
- `read()`: Read a number of bytes from a file handle into a buffer
- `write()`: Write a number of bytes from a buffer to the file handle
- `close()`: Close the file handle
- `lseek()`: Change position in the file handle
- `access()`: Check access rights based on real user ID (more later)

# Pseudo filesystems `proc` and `sys`

- Files in `/proc` and `/sys` are “pseudo-files”
- Those files provide reading or writing access to OS parameters

# Pseudo filesystems `proc` and `sys`

- Files in `/proc` and `/sys` are “pseudo-files”
- Those files provide reading or writing access to OS parameters
- Examples:
  - `cat /proc/cpuinfo`: Shows all kind of information about the CPUs of the system
  - `cat /proc/meminfo`: Shows all kind of information about the memory of the system

# Pseudo filesystems `proc` and `sys`

- Files in `/proc` and `/sys` are "pseudo-files"
- Those files provide reading or writing access to OS parameters
- Examples:
  - `cat /proc/cpuinfo`: Shows all kind of information about the CPUs of the system
  - `cat /proc/meminfo`: Shows all kind of information about the memory of the system
  - `echo 1 > /proc/sys/net/ipv4/ip_forward`: Enable IP forwarding
  - `echo powersave > /sys/.../cpu0/cpufreq/scaling_governor`: Switch CPU0 to "powersave" mode

# Pseudo filesystems `proc` and `sys`

- Files in `/proc` and `/sys` are "pseudo-files"
- Those files provide reading or writing access to OS parameters
- Examples:
  - `cat /proc/cpuinfo`: Shows all kind of information about the CPUs of the system
  - `cat /proc/meminfo`: Shows all kind of information about the memory of the system
  - `echo 1 > /proc/sys/net/ipv4/ip_forward`: Enable IP forwarding
  - `echo powersave > /sys/.../cpu0/cpufreq/scaling_governor`: Switch CPU0 to "powersave" mode
- Important for access control: reading/writing those parameters is implemented through operations on (pseudo-)files

# Device files

- Hardware devices are represented as files in `/dev/`
- Examples:
  - `/dev/sda`: First hard drive
  - `/dev/sda1`: First partition on first hard drive
  - `/dev/tty*`: Serial devices and terminals
  - `/dev/input/*`: Input devices
  - `/dev/zero`: Pseudo-devices that prints zeros
  - `/dev/random`: Pseudo-devices that prints random bytes

# Device files

- Hardware devices are represented as files in `/dev/`
- Examples:
  - `/dev/sda`: First hard drive
  - `/dev/sda1`: First partition on first hard drive
  - `/dev/tty*`: Serial devices and terminals
  - `/dev/input/*`: Input devices
  - `/dev/zero`: Pseudo-devices that prints zeros
  - `/dev/random`: Pseudo-devices that prints random bytes
- Generally be very careful when writing to device files
- `dd if=/dev/zero of=/dev/sda` overwrites your whole hard drive with zeros

# Device files

- Hardware devices are represented as files in `/dev/`
- Examples:
  - `/dev/sda`: First hard drive
  - `/dev/sda1`: First partition on first hard drive
  - `/dev/tty*`: Serial devices and terminals
  - `/dev/input/*`: Input devices
  - `/dev/zero`: Pseudo-devices that prints zeros
  - `/dev/random`: Pseudo-devices that prints random bytes
- Generally be very careful when writing to device files
- `dd if=/dev/zero of=/dev/sda` overwrites your whole hard drive with zeros
- Again, important for access control: accessing (hardware) devices is implemented through operations on (device-)files

# Symbolic links and pipes

- A *symbolic link* is a special file that “links” to another file
- Accessing a symbolic link really accesses the file it points to
- Create a symbolic link to /home/peter/teaching/ with name /home/peter/ru:

  ```
  ln -s /home/peter/teaching /home/peter/ru
  ```

# Symbolic links and pipes

- A *symbolic link* is a special file that “links” to another file
- Accessing a symbolic link really accesses the file it points to
- Create a symbolic link to /home/peter/teaching/ with name /home/peter/ru:

  `ln -s /home/peter/teaching /home/peter/ru`

- Can also create a *hard link*:

  `ln /home/peter/teaching /home/peter/ru`

- Soft links don’t get updated when the target is moved
- Hard links always point to the target

# Symbolic links and pipes

- A *symbolic link* is a special file that "links" to another file
- Accessing a symbolic link really accesses the file it points to
- Create a symbolic link to /home/peter/teaching/ with name /home/peter/ru:

  ```
  ln -s /home/peter/teaching /home/peter/ru
  ```

- Can also create a *hard link*:

  ```
  ln /home/peter/teaching /home/peter/ru
  ```

- Soft links don't get updated when the target is moved
- Hard links always point to the target
- Access is again handled through file handles, need to be careful with permissions

# Symbolic links and pipes

- A *symbolic link* is a special file that "links" to another file
- Accessing a symbolic link really accesses the file it points to
- Create a symbolic link to /home/peter/teaching/ with name /home/peter/ru:

  `ln -s /home/peter/teaching /home/peter/ru`

- Can also create a *hard link*:

  `ln /home/peter/teaching /home/peter/ru`

- Soft links don't get updated when the target is moved
- Hard links always point to the target
- Access is again handled through file handles, need to be careful with permissions
- Pipes for inter-process communication are also implemented through file handles

# Environment variables

- One might think that data flow between processes can only happen through files
- Process A writes a file, process B reads the file

# Environment variables

- One might think that data flow between processes can only happen through files
- Process A writes a file, process B reads the file
- Other way of communicating: environment variables
- Process A can set an environment variable, process B can read it

# Environment variables

- One might think that data flow between processes can only happen through files
- Process A writes a file, process B reads the file
- Other way of communicating: environment variables
- Process A can set an environment variable, process B can read it
- Set an environment variable through

  ```
  export MYVAR=myvalue
  ```

- Show all currently defined environment variables: `export`

# Environment variables

- One might think that data flow between processes can only happen through files
- Process A writes a file, process B reads the file
- Other way of communicating: environment variables
- Process A can set an environment variable, process B can read it
- Set an environment variable through

  `export MYVAR=myvalue`

- Show all currently defined environment variables: `export`
- Important system-wide variables:
  - `PATH`: colon-separated list of directories to search for programs
  - `LD_LIBRARY_PATH`: colon-separated list of directories to search for libraries
  - `IFS`: "Internal Field Separator", character to be used to separate fields in a list (more later)

# MAC and DAC

## Protection system

A *protection system* consists of a *protection state*, which describes what operations subjects (processes) may perform on objects (files) together with a set of *protection state operations* that enable modification of the state.

# MAC and DAC

## Protection system

A *protection system* consists of a *protection state*, which describes what operations subjects (processes) may perform on objects (files) together with a set of *protection state operations* that enable modification of the state.

## Mandatory Access Control

A system implements *mandatory access control* (MAC) if the protection state can only be modified by trusted administrators via trusted software.

# MAC and DAC

## Protection system

A *protection system* consists of a *protection state*, which describes what operations subjects (processes) may perform on objects (files) together with a set of *protection state operations* that enable modification of the state.

## Mandatory Access Control

A system implements *mandatory access control* (MAC) if the protection state can only be modified by trusted administrators via trusted software.

## Discretionary Access Control

A system implements *discretionary access control* (DAC) if the protection state can be modified by untrusted users. The protection of a user's files is then "at the discretion of the user".

# Access Matrix

An *access matrix* is a set of subjects $S$, a set of objects $O$, a set of operations $X$ and a function $op : S \times O \rightarrow \mathcal{P}(X)$. Given $s \in S$ and $o \in O$, the function $op$ returns the set of operations that $s$ is allowed to perform on $o$.

# Access Matrix

| | File 1 | File 2 | File 3 | File 4 |
|---|---|---|---|---|
| Process 1 | read | read | read,write | |
| Process 2 | | read | | |
| Process 3 | read,write | read | | |

# Access Matrix

| | File 1 | File 2 | File 3 | File 4 |
|---|---|---|---|---|
| Process 1 | read | read | read,write | |
| Process 2 | | read | | |
| Process 3 | read,write | read | | |

- When a user creates a file, she adds a column to the table
- Adding a column means modifying the protection state
- The access-matrix model leads to a DAC system

# UNIX/Linux protection model

- *Trusted code base* (TCB) of Linux is all code running in kernel space and several processes running with `root` permissions, e.g.:
  - `init` process
  - `login` (user authentication)
  - network services
- Goal: protect users' processes from each other and the TCB from all user processes

# UNIX/Linux protection model: subjects

- Each process has associated three user IDs:
  - Real user ID
  - Effective user ID
  - Saved user ID

# UNIX/Linux protection model: subjects

- Each process has associated three user IDs:
  - Real user ID
  - Effective user ID
  - Saved user ID
- Each process also has associated a set of *group IDs*
- The groups of all users are defined in `/etc/group`
- Each user has a primary group defined in `/etc/passwd`
- When you are logged in, you can see your groups with the command `groups`

# UNIX/Linux protection model: objects

- Each object (file) has
  - an owner (user) and owner permissions
  - a group and group permissions
  - other permissions

# UNIX/Linux protection model: objects

- Each object (file) has
  - an owner (user) and owner permissions
  - a group and group permissions
  - other permissions
- Permissions on a file are read (**r**), write (**w**) and execute (**x**)
- Typically write permissions as 9 bits: $\underbrace{\text{rwx}}_{owner}\ \underbrace{\text{rwx}}_{group}\ \underbrace{\text{rwx}}_{other}$

# UNIX/Linux protection model: objects

- Each object (file) has
  - an owner (user) and owner permissions
  - a group and group permissions
  - other permissions
- Permissions on a file are read (**r**), write (**w**) and execute (**x**)
- Typically write permissions as 9 bits: $\underbrace{\text{rwx}}_{owner}\ \underbrace{\text{rwx}}_{group}\ \underbrace{\text{rwx}}_{other}$
- Convenient way of writing this: 3 numbers from 0–7, e.g.:
  - 750: owner may read, write, and execute; group may read and execute, others may nothing
  - 644: owner may read and write; group and others may read

# UNIX/Linux protection model: objects

- Each object (file) has
  - an owner (user) and owner permissions
  - a group and group permissions
  - other permissions
- Permissions on a file are read (**r**), write (**w**) and execute (**x**)
- Typically write permissions as 9 bits: $\underbrace{\text{rwx}}_{owner}\ \underbrace{\text{rwx}}_{group}\ \underbrace{\text{rwx}}_{other}$
- Convenient way of writing this: 3 numbers from 0–7, e.g.:
  - 750: owner may read, write, and execute; group may read and execute, others may nothing
  - 644: owner may read and write; group and others may read
- Command `ls -l` shows files with corresponding permissions, e.g.

```
peter@tyrion:/etc$ ls -l passwd shadow
-rw-r--r-- 1 root root   2217 Nov 16 18:13 passwd
-rw-r----- 1 root shadow 1454 Nov 16 18:13 shadow
```

# UNIX/Linux protection model: matching

- When a process wants to access a file, check the following
  1. Does the effective user ID of the process match the owner of the file? If so, use the owner permissions.
  2. Does one of the group IDs of the process match the group of the file? If so, use the group permissions.
  3. Otherwise, use the “other” permissions

# UNIX/Linux protection model: matching

- When a process wants to access a file, check the following
  1. Does the effective user ID of the process match the owner of the file? If so, use the owner permissions.
  2. Does one of the group IDs of the process match the group of the file? If so, use the group permissions.
  3. Otherwise, use the "other" permissions
- Note: if the owner matches, the group permissions don't matter.

# UNIX/Linux protection model: matching

- When a process wants to access a file, check the following
  1. Does the effective user ID of the process match the owner of the file? If so, use the owner permissions.
  2. Does one of the group IDs of the process match the group of the file? If so, use the group permissions.
  3. Otherwise, use the “other” permissions
- Note: if the owner matches, the group permissions don’t matter.

## Directory permissions

- `read`: Can see content (files and subdirectories) of the directory

# UNIX/Linux protection model: matching

- When a process wants to access a file, check the following
  1. Does the effective user ID of the process match the owner of the file? If so, use the owner permissions.
  2. Does one of the group IDs of the process match the group of the file? If so, use the group permissions.
  3. Otherwise, use the “other” permissions
- Note: if the owner matches, the group permissions don’t matter.

## Directory permissions

- `read`: Can see content (files and subdirectories) of the directory
- `write`: Can rename and delete content of the directory and create new content

# UNIX/Linux protection model: matching

- When a process wants to access a file, check the following
  1. Does the effective user ID of the process match the owner of the file? If so, use the owner permissions.
  2. Does one of the group IDs of the process match the group of the file? If so, use the group permissions.
  3. Otherwise, use the "other" permissions
- Note: if the owner matches, the group permissions don't matter.

## Directory permissions

- `read`: Can see content (files and subdirectories) of the directory
- `write`: Can rename and delete content of the directory and create new content
- `execute`: Can traverse the directory (`cd` into or across the directory)

# chown, chmod and umask

- chown changes owner and group of a file
- Example: `chown veelasha:dialout test.txt` changes
  - the owner of test.txt to veelasha and
  - the group of test.txt to dialout
- Only root can change ownership; owner can change group to any group he's member of

# chown, chmod and umask

- `chown` changes owner and group of a file
- Example: `chown veelasha:dialout test.txt` changes
  - the owner of `test.txt` to `veelasha` and
  - the group of `test.txt` to `dialout`
- Only root can change ownership; owner can change group to any group he's member of
- `chmod` changes permissions of a file, e.g.:
  - `chmod g+w`: grant write permissions to group
  - `chmod o-x`: remove execute permissions from other
  - `chmod a+rw`: grant read and write permissions to owner, group, and other
  - `chmod 640`: set permissions to `rw-r-----`

# chown, chmod and umask

- chown changes owner and group of a file
- Example: chown veelasha:dialout test.txt changes
  - the owner of test.txt to veelasha and
  - the group of test.txt to dialout
- Only root can change ownership; owner can change group to any group he's member of
- chmod changes permissions of a file, e.g.:
  - chmod g+w: grant write permissions to group
  - chmod o-x: remove execute permissions from other
  - chmod a+rw: grant read and write permissions to owner, group, and other
  - chmod 640: set permissions to rw-r-----
- Default permissions for files are 666 and for directories 777
- umask influences default permissions
- The umask is subtracted from permissions
- Example: a umask of 022 removes write permissions for group and other by default

# The setuid bit

- Sometimes users need to have access to privileged resources
- UNIX/Linux solution: additional *setuid (suid) bit* in file permissions
- Run program with permissions of *owner* instead of user starting it

# The setuid bit

- Sometimes users need to have access to privileged resources
- UNIX/Linux solution: additional *setuid (suid) bit* in file permissions
- Run program with permissions of *owner* instead of user starting it
- Set suid bit with `chmod u+s` or, e.g., `chmod 4755`

# The setuid bit

- Sometimes users need to have access to privileged resources
- UNIX/Linux solution: additional *setuid (suid) bit* in file permissions
- Run program with permissions of *owner* instead of user starting it
- Set suid bit with `chmod u+s` or, e.g., `chmod 4755`
- User IDs of a suid program:
  - Real user ID: ID of the user starting the program
  - Effective user ID: ID of the owner
  - Saved user ID: set to effective user ID at the beginning

# The setuid bit

- Sometimes users need to have access to privileged resources
- UNIX/Linux solution: additional *setuid (suid) bit* in file permissions
- Run program with permissions of *owner* instead of user starting it
- Set suid bit with `chmod u+s` or, e.g., `chmod 4755`
- User IDs of a suid program:
  - Real user ID: ID of the user starting the program
  - Effective user ID: ID of the owner
  - Saved user ID: set to effective user ID at the beginning
- Most important application: setuid root
- Setuid root process can drop privileges (effective ID)
- Can regain root rights as long as saved ID is still 0!

# The setgid and sticky bit

## setgid bit

- When set on executable file: use effective group ID for process

# The setgid and sticky bit

## setgid bit

- When set on executable file: use effective group ID for process
- Different meaning for directories: files created within this directory inherit the group ID
- Similar mechanism for suid on directories on a few systems (not on Linux)

# The setgid and sticky bit

## setgid bit

- When set on executable file: use effective group ID for process
- Different meaning for directories: files created within this directory inherit the group ID
- Similar mechanism for suid on directories on a few systems (not on Linux)
- Set setgid bit with `chmod g+s` or, e.g., `chmod 2777`

# The setgid and sticky bit

## setgid bit

- When set on executable file: use effective group ID for process
- Different meaning for directories: files created within this directory inherit the group ID
- Similar mechanism for suid on directories on a few systems (not on Linux)
- Set setgid bit with `chmod g+s` or, e.g., `chmod 2777`

## Sticky bit

- Another “special” permission bit is the *sticky bit*
- On directories: allow only owner of contained files to rename or delete the file
- Important, for example, for `/tmp/`
- On executables: keep in swap space (faster loading)
- Not really used anymore today
- Set sticky bit with `chmod +t`

# setuid example: `su`

- Most prominent example of setuid-root program: `su`
- `su` can stand for "switch user" or "superuser"
- Without any argument, become `root`
- Can provide other username as argument

# setuid example: su

- Most prominent example of setuid-root program: su
- su can stand for "switch user" or "superuser"
- Without any argument, become root
- Can provide other username as argument
- Authentication uses PAM, typical (piece of) /etc/pam.d/su:

```
auth        sufficient pam_rootok.so
session     required   pam_limits.so
auth        required   pam_unix.so
```

# setuid example: su

- Most prominent example of setuid-root program: `su`
- `su` can stand for "switch user" or "superuser"
- Without any argument, become `root`
- Can provide other username as argument
- Authentication uses PAM, typical (piece of) `/etc/pam.d/su`:

```
auth       sufficient pam_rootok.so
session    required   pam_limits.so
auth       required   pam_unix.so
```

- Other prominent example: `passwd` (needs write access to `/etc/shadow`)
- Again, authenticate against PAM before doing anything

# sudo

- `su` requires users to authenticate as root
- `sudo` allows users to authenticate as themselves and run commands with root privileges
- `sudo` also uses suid root and PAM
- Configuration of users and permitted commands in `/etc/sudoers`

# sudo

- `su` requires users to authenticate as root
- `sudo` allows users to authenticate as themselves and run commands with root privileges
- `sudo` also uses suid root and PAM
- Configuration of users and permitted commands in `/etc/sudoers`
- Some Linux Distributions (Ubuntu) disable the `root` password
- Instead use the following rule in `etc/sudoers`:

  `%sudo ALL=(ALL:ALL) ALL`

# sudo

- `su` requires users to authenticate as root
- `sudo` allows users to authenticate as themselves and run commands with root privileges
- `sudo` also uses suid root and PAM
- Configuration of users and permitted commands in `/etc/sudoers`
- Some Linux Distributions (Ubuntu) disable the `root` password
- Instead use the following rule in `etc/sudoers`:

  `%sudo ALL=(ALL:ALL) ALL`
- Allows members of the group `sudo` to run any program as root
- With this rule, run `sudo su` to obtain a root shell

# Privilege escalation

- Attack that expands attacker's privileges is called *privilege escalation*
- Two types of privilege escalation:
  - horizontal: obtain privileges of another un-privileged user
  - vertical: obtain privileges of root (or the kernel), "privilege elevation"

# Privilege escalation

- Attack that expands attacker's privileges is called *privilege escalation*
- Two types of privilege escalation:
  - horizontal: obtain privileges of another un-privileged user
  - vertical: obtain privileges of root (or the kernel), "privilege elevation"
- Typicall enabled by bugs in privileged software:
  - Bugs in the kernel
  - Bugs in how `root` programs process user-provided input
  - Bugs in suid-root programs (escape intended functionality)

# Privilege escalation

- Attack that expands attacker's privileges is called *privilege escalation*
- Two types of privilege escalation:
  - horizontal: obtain privileges of another un-privileged user
  - vertical: obtain privileges of root (or the kernel), "privilege elevation"
- Typicall enabled by bugs in privileged software:
  - Bugs in the kernel
  - Bugs in how `root` programs process user-provided input
  - Bugs in suid-root programs (escape intended functionality)
- An exploit that lets an unprivileged (logged in, local) user gain root rights is called *local root exploit*

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`
- Attacker proceeds as follows:
  1. `export PATH=.:$PATH`

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`
- Attacker proceeds as follows:
  1. `export PATH=.:$PATH`
  2. `export IFS=/`

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`
- Attacker proceeds as follows:
  1. `export PATH=.:$PATH`
  2. `export IFS=/`
  3. Create executable file `./bin` containing:

     ```
     cp /bin/sh ./myrootshell
     chown root:root ./myrootshell
     chmod u+s ./myrootshell
     ```

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`
- Attacker proceeds as follows:
  1. `export PATH=.:$PATH`
  2. `export IFS=/`
  3. Create executable file `./bin` containing:

     ```
     cp /bin/sh ./myrootshell
     chown root:root ./myrootshell
     chmod u+s ./myrootshell
     ```

  4. Run the suid program `stupid`

# Using `system()` with suid

- The `system()` function runs another program in the shell
- Uses the `fork()` and the `execve()` system calls
- **Never** use `system()` in a suid program!
- Example: suid program `stupid` contains `system("/bin/date")`
- Attacker proceeds as follows:
  1. `export PATH=.:$PATH`
  2. `export IFS=/`
  3. Create executable file `./bin` containing:
     ```
     cp /bin/sh ./myrootshell
     chown root:root ./myrootshell
     chmod u+s ./myrootshell
     ```
  4. Run the suid program `stupid`
- `stupid` launches a shell, which is handed `/bin/date`
- Shell looks at variable `IFS` to parse this string
- Shell calls program `bin` with argument `date`

# IFS and LD_LIBRARY_PATH

- Attack against `system("/bin/date")` does not work anymore
- `IFS` environment variable is no longer inherited by shells

# IFS and LD_LIBRARY_PATH

- Attack against system("/bin/date") does not work anymore
- IFS environment variable is no longer inherited by shells
- LD_LIBRARY_PATH is not inherited for programs with setuid bit set

# IFS and LD_LIBRARY_PATH

- Attack against `system("/bin/date")` does not work anymore
- `IFS` environment variable is no longer inherited by shells
- `LD_LIBRARY_PATH` is not inherited for programs with setuid bit set
- `PATH` variable is still inherited
- Custom variables are still inherited

# IFS and LD_LIBRARY_PATH

- Attack against `system("/bin/date")` does not work anymore
- `IFS` environment variable is no longer inherited by shells
- `LD_LIBRARY_PATH` is not inherited for programs with setuid bit set
- `PATH` variable is still inherited
- Custom variables are still inherited
- Can try all this easily with a C program using `getenv`

# IFS and LD_LIBRARY_PATH

- Attack against `system("/bin/date")` does not work anymore
- `IFS` environment variable is no longer inherited by shells
- `LD_LIBRARY_PATH` is not inherited for programs with setuid bit set
- `PATH` variable is still inherited
- Custom variables are still inherited
- Can try all this easily with a C program using `getenv`
- Cannot try this with a shell script
- Shell scripts won't execute setuid (even if you set the bit)

# Shellshock

- Environment variables can be dangerous because they allow (potentially unintended) data flow
- Even worse if environment variables are badly parsed: http://digg.com/video/the-shellshock-bug-explained-in-about-four-minutes

# More Shellshock background

- The bash is not just a command line but also a programming language
- We can define functions: `hello() { echo "Hello World"; }`
- We can also `export` functions with `export -f`

# More Shellshock background

- The bash is not just a command line but also a programming language
- We can define functions: `hello() { echo "Hello World"; }`
- We can also `export` functions with `export -f`
- Environment variables do not support functions, just strings
- The newly launched bash looks for variables that “look like a function”
- Parsing things that “look like a function” goes wrong

# Shellshock test

```
env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
```

# Access control lists

- User/Group/All model is not always flexible enough
- Want to enable arbitrary access permissions
- Solution: Access Control Lists (ACLs)

# Access control lists

- User/Group/All model is not always flexible enough
- Want to enable arbitrary access permissions
- Solution: Access Control Lists (ACLs)
- Grant permissions to arbitrary users and groups
- Needs support from the file system
- Mount with option `acl`, for example:

```
mount -o remount,acl /
```

# Access control lists

- User/Group/All model is not always flexible enough
- Want to enable arbitrary access permissions
- Solution: Access Control Lists (ACLs)
- Grant permissions to arbitrary users and groups
- Needs support from the file system
- Mount with option `acl`, for example:

  ```
  mount -o remount,acl /
  ```

- Set ACL entries with the program `setfacl` (set file access control lists)
- Read ACL entries with `getfacl` (get file access control lists)
- Note: `ls -l` will not show ACLs, only a '+' to indicate that "there's more"

# Linux ACL examples

- Grant user `veelasha` read,write execute rights on file `test.txt`:

  ```
  setfacl -m user:veelasha:rwx test.txt
  ```

# Linux ACL examples

- Grant user `veelasha` read,write execute rights on file `test.txt`:

  `setfacl -m user:veelasha:rwx test.txt`

- Remove all rights for user `veelasha` on file `test.txt`:

  `setfacl -x user:veelasha test.txt`

# Linux ACL examples

- Grant user `veelasha` read,write execute rights on file `test.txt`:

  `setfacl -m user:veelasha:rwx test.txt`

- Remove all rights for user `veelasha` on file `test.txt`:

  `setfacl -x user:veelasha test.txt`

- Grant read and execute rights for members of the group `dialout`:

  `setfacl -m group:dialout:r-x test.txt`

# Linux ACL examples

- Grant user `veelasha` read,write execute rights on file `test.txt`:
  `setfacl -m user:veelasha:rwx test.txt`
- Remove all rights for user `veelasha` on file `test.txt`:
  `setfacl -x user:veelasha test.txt`
- Grant read and execute rights for members of the group `dialout`:
  `setfacl -m group:dialout:r-x test.txt`
- Read and set permissions for `test.txt` from file `test.perm`:
  `setfacl -M test.perm test.txt`

# UNIX weaknesses: assuming benign processes

- UNIX and Linux are built on the assumption that user processes behave benignly
- A malicious process can easily violate a user's security goals
- Mainly two ways why processes may be malicious:
  - user accidently runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)

# UNIX weaknesses: assuming benign processes

- UNIX and Linux are built on the assumption that user processes behave benignly
- A malicious process can easily violate a user's security goals
- Mainly two ways why processes may be malicious:
  - user accidently runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)
- Ideal situation: OS enforces security:
  - Clearly defined security goals (confidentiality, integrity)
  - All software outside the TBC can be arbitrarily malicious
  - OS still enforces the security goals

# UNIX weaknesses: assuming benign processes

- UNIX and Linux are built on the assumption that user processes behave benignly
- A malicious process can easily violate a user's security goals
- Mainly two ways why processes may be malicious:
  - user accidently runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)
- Ideal situation: OS enforces security:
  - Clearly defined security goals (confidentiality, integrity)
  - All software outside the TBC can be arbitrarily malicious
  - OS still enforces the security goals
- No current mainstream OS achieves this goal
- Requires mandatory access control

# UNIX weaknesses: TOCTTOU

- Problem if there is a time gap between checking permissions and executing operation
- This is called *time of check to time of use* (TOCTTOU or TOCTOU)

# UNIX weaknesses: TOCTTOU

- Problem if there is a time gap between checking permissions and executing operation
- This is called *time of check to time of use* (TOCTTOU or TOCTOU)
- Example: use `access()` syscall in suid-root program to check rights against *real* user ID:

```
if (access("file", W_OK) != 0) {
   exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

# UNIX weaknesses: TOCTTOU

- Problem if there is a time gap between checking permissions and executing operation
- This is called *time of check to time of use* (TOCTTOU or TOCTOU)
- Example: use `access()` syscall in suid-root program to check rights against *real* user ID:

```
if (access("file", W_OK) != 0) {
   exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Attacker attempts to run `symlink("/etc/shadow", "file");` between `access()` and `open()`

# UNIX weaknesses: TOCTTOU

- Problem if there is a time gap between checking permissions and executing operation
- This is called *time of check to time of use* (TOCTTOU or TOCTOU)
- Example: use `access()` syscall in suid-root program to check rights against *real* user ID:

```
if (access("file", W_OK) != 0) {
   exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Attacker attempts to run `symlink("/etc/shadow", "file");` between `access()` and `open()`
- This is an example for a *race condition*
- Generally, a *race condition bug* is a bug where software behaviour depends on uncontrollable timing behavior in an unintended way