OS Security

Authentication and Authorization

Radboud University, Nijmegen, The Netherlands

Winter 2016/2017

What does an OS do?

Definition

An *operating system (OS)* is a computer program that manages access of processes (programs) to shared resources.

What does an OS do?

Definition

An *operating system (OS)* is a computer program that manages access of processes (programs) to shared resources.

Examples of shared resources

- Memory
- Input and Output (I/O) including
  - Files on the hard drive
  - Network
- Computation cycles on the processor(s)
- Peripheral hardware (keyboard, screen, ...)

What does that mean for security?

- Operating system needs to decide whether processes are allowed to perform certain operations
- Obvious security disasters:
  - One process reading the memory of another process
  - A process reading a "secret" file
  - A process preventing other processes from operating
  - One process reading (keyboard) input meant for another process

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Is user `peter` allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Is user `peter` allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Need to map between a user (human) and a certain operation

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Is user `peter` allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Need to map between a user (human) and a certain operation

Definition

Authentication is the means by which it is determined that a particular entity (typically a human) intends to perform a given operation.

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Is user `peter` allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Need to map between a user (human) and a certain operation

Definition
Authentication is the means by which it is determined that a particular entity (typically a human) intends to perform a given operation.

- Typically perform *user authentication* as a login procedure
- Start a shell mapped to the logged-in user
- A shell is (basically) an interface to run other programs
- All programs run from this shell are mapped to the logged-in user

Wait, what about users?

- Is the process with ID 4321 allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Is user `peter` allowed to read the file `/home/peter/os-security/exam-jan-2017.pdf`?
- Need to map between a user (human) and a certain operation

Definition

Authentication is the means by which it is determined that a particular entity (typically a human) intends to perform a given operation.

- Typically perform *user authentication* as a login procedure
- Start a shell mapped to the logged-in user
- A shell is (basically) an interface to run other programs
- All programs run from this shell are mapped to the logged-in user
- Worst-case authentication failure: impersonation

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files
- `root` may change permissions on all files

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files
- `root` may change permissions on all files
- `root` may bind programs to network sockets with port number smaller than 1024

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files
- `root` may change permissions on all files
- `root` may bind programs to network sockets with port number smaller than 1024
- `root` may "impersonate" any other user
- A process belonging to `root` may change its user ID to that of another user
- Once a process has changed from user ID 0 to another user ID, there is no way back

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files
- `root` may change permissions on all files
- `root` may bind programs to network sockets with port number smaller than 1024
- `root` may "impersonate" any other user
- A process belonging to `root` may change its user ID to that of another user
- Once a process has changed from user ID 0 to another user ID, there is no way back
- There are still certain actions that a program run by `root` cannot do (more next lecture)

The user `root`

- UNIX and Linux have a special *superuser* called `root`
- The user ID of `root` is always 0
- `root` may access all files
- `root` may change permissions on all files
- `root` may bind programs to network sockets with port number smaller than 1024
- `root` may "impersonate" any other user
- A process belonging to `root` may change its user ID to that of another user
- Once a process has changed from user ID 0 to another user ID, there is no way back
- There are still certain actions that a program run by `root` cannot do (more next lecture)
- **Security nightmare:** an attacker who gets `root` access

Classical UNIX/Linux authentication

- Authentication by “what you know”
- Init process starts `login` (runs as root)

Classical UNIX/Linux authentication

- Authentication by "what you know"
- Init process starts `login` (runs as root)
- `login` prompts for username and password
- Correct password: `login` changes to new user and executes a shell

Classical UNIX/Linux authentication

- Authentication by "what you know"
- Init process starts `login` (runs as root)
- `login` prompts for username and password
- Correct password: `login` changes to new user and executes a shell
- Comparison of *password hash* against info stored in `/etc/shadow` (originally `/etc/passwd`)

Password hashing algorithms

- Traditionally Linux used `crypt` for password hashing
- Truncate the password to 8 characters, 7 bits each
- Encrypt the all-zero string with modified DES with this 56-bit key
- Iterate encryption for 25 times (later: up to $2^{24} - 1$)
- Incorporate a 12-bit (later: 24-bit) salt

Password hashing algorithms

- Traditionally Linux used `crypt` for password hashing
- Truncate the password to 8 characters, 7 bits each
- Encrypt the all-zero string with modified DES with this 56-bit key
- Iterate encryption for 25 times (later: up to $2^{24} - 1$)
- Incorporate a 12-bit (later: 24-bit) salt
- Use *modified* DES to prevent attacks with DES hardware
- Originally computing the hash cost ≈ 1 second
- Too weak nowadays to offer strong protection

Password hashing algorithms

- Traditionally Linux used `crypt` for password hashing
- Truncate the password to 8 characters, 7 bits each
- Encrypt the all-zero string with modified DES with this 56-bit key
- Iterate encryption for 25 times (later: up to $2^{24} - 1$)
- Incorporate a 12-bit (later: 24-bit) salt
- Use *modified* DES to prevent attacks with DES hardware
- Originally computing the hash cost ≈ 1 second
- Too weak nowadays to offer strong protection
- Successors: MD5, `bcrypt` (based on Blowfish), SHA-2

Password hashing algorithms

- Traditionally Linux used `crypt` for password hashing
- Truncate the password to 8 characters, 7 bits each
- Encrypt the all-zero string with modified DES with this 56-bit key
- Iterate encryption for 25 times (later: up to $2^{24} - 1$)
- Incorporate a 12-bit (later: 24-bit) salt
- Use *modified* DES to prevent attacks with DES hardware
- Originally computing the hash cost ≈ 1 second
- Too weak nowadays to offer strong protection
- Successors: MD5, `bcrypt` (based on Blowfish), SHA-2
- Password hash string indicates which algorithm to use:
  - **1**: MD5;
  - **$2a$**, **$2b$**, **$2x$**, **$2y$**: variants of `bcrypt`
  - **5**: SHA-256; **6**: SHA-512

Password hashing algorithms

- Traditionally Linux used `crypt` for password hashing
- Truncate the password to 8 characters, 7 bits each
- Encrypt the all-zero string with modified DES with this 56-bit key
- Iterate encryption for 25 times (later: up to $2^{24} - 1$)
- Incorporate a 12-bit (later: 24-bit) salt
- Use *modified* DES to prevent attacks with DES hardware
- Originally computing the hash cost ≈ 1 second
- Too weak nowadays to offer strong protection
- Successors: MD5, `bcrypt` (based on Blowfish), SHA-2
- Password hash string indicates which algorithm to use:
  - **1**: MD5;
  - **$2a$**, **$2b$**, **$2x$**, **$2y$**: variants of `bcrypt`
  - **5**: SHA-256; **6**: SHA-512
- Better algorithm through https://password-hashing.net/
- Winner announced on Nov 2, 2015: ARGON2

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
  1. Restrict password to 14 characters

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
    1. Restrict password to 14 characters
    2. Convert password to all-uppercase

How about Windows?

- Traditionally, Windows uses the LM hash (for “LanMan hash” or “LAN manager hash”)
- Algorithm for LM hash:
  1. Restrict password to 14 characters
  2. Convert password to all-uppercase
  3. Pad to 14 bytes

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
  1. Restrict password to 14 characters
  2. Convert password to all-uppercase
  3. Pad to 14 bytes
  4. Split into two 7-byte halves

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
  1. Restrict password to 14 characters
  2. Convert password to all-uppercase
  3. Pad to 14 bytes
  4. Split into two 7-byte halves
  5. Use each of the halves as a DES key

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
  1. Restrict password to 14 characters
  2. Convert password to all-uppercase
  3. Pad to 14 bytes
  4. Split into two 7-byte halves
  5. Use each of the halves as a DES key
  6. Encrypt the fixed ASCII string `KGS!@#$%`

How about Windows?

- Traditionally, Windows uses the LM hash (for "LanMan hash" or "LAN manager hash")
- Algorithm for LM hash:
  1. Restrict password to 14 characters
  2. Convert password to all-uppercase
  3. Pad to 14 bytes
  4. Split into two 7-byte halves
  5. Use each of the halves as a DES key
  6. Encrypt the fixed ASCII string `KGS!@#$%`
  7. Concatenate the two ciphertexts to obtain the LM hash

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords
- Can crack the halves independently: 2^{46} for each half

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords
- Can crack the halves independently: 2^{46} for each half
- All characters converted to upper case: 2^{43} for each half

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords
- Can crack the halves independently: 2^{46} for each half
- All characters converted to upper case: 2^{43} for each half
- No salt, rainbow tables are feasible

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords
- Can crack the halves independently: 2^{46} for each half
- All characters converted to upper case: 2^{43} for each half
- No salt, rainbow tables are feasible
- Passwords shorter than 8 characters produce hash ending in `0xAAD3B435B51404EE`

LM Hash weaknesses

- 14 printable ASCII characters give $\approx 2^{92}$ passwords
- Can crack the halves independently: 2^{46} for each half
- All characters converted to upper case: 2^{43} for each half
- No salt, rainbow tables are feasible
- Passwords shorter than 8 characters produce hash ending in `0xAAD3B435B51404EE`
- Cracking LM hashes is fairly easy, there are even online services, e.g., http://rainbowtables.it64.com/

NT hashes

- LM hash weaknesses were addressed by NT hash (or NTLM)
- NTLMv1 uses MD4 to hash passwords
- NTLMv2 uses MD5 to hash passwords
- Passwords are still unsalted

NT hashes

- LM hash weaknesses were addressed by NT hash (or NTLM)
- NTLMv1 uses MD4 to hash passwords
- NTLMv2 uses MD5 to hash passwords
- Passwords are still unsalted
- Until Windows XP, LM hashes were still enabled by default for backwards compatibility

NT hashes

- LM hash weaknesses were addressed by NT hash (or NTLM)
- NTLMv1 uses MD4 to hash passwords
- NTLMv2 uses MD5 to hash passwords
- Passwords are still unsalted
- Until Windows XP, LM hashes were still enabled by default for backwards compatibility
- Today, Windows uses multiple different approaches for passwords

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie "Spaceballs"...?

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie “Spaceballs”. . . ?
- Most common passwords in 2014 (SplashData list):

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie “Spaceballs”…?
- Most common passwords in 2014 (SplashData list):
  - Place 3: **12345**

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie "Spaceballs"...?
- Most common passwords in 2014 (SplashData list):
  - Place 3: **12345**
  - Place 2: **password**

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie "Spaceballs"...?
- Most common passwords in 2014 (SplashData list):
  - Place 3: **12345**
  - Place 2: **password**
  - Place 1: **123456**

Weak passwords

- Largest problem with passwords: weak passwords
- Remember 1987 Mel Brooks movie “Spaceballs”...?
- Most common passwords in 2014 (SplashData list):
  - Place 3: **12345**
  - Place 2: **password**
  - Place 1: **123456**
- Exercises in 1st semester course include breaking (unsalted) hash of a 7-character random password.
- Some students typically manage to do that in a week!

Authentication by “what you have”

- Very common in the “physical world”, e.g., keys
- Digital world: Smart cards, USB tokens
- Private keys (e.g., for SSH)

Authentication by “what you have”

- Very common in the “physical world”, e.g., keys
- Digital world: Smart cards, USB tokens
- Private keys (e.g., for SSH)
- Can easily combine with password, e.g. on SSH private keys

Authentication by “what you have”

- Very common in the “physical world”, e.g., keys
- Digital world: Smart cards, USB tokens
- Private keys (e.g., for SSH)
- Can easily combine with password, e.g. on SSH private keys

Attacks and countermeasures

- **Stealing (or finding):** Protect possession

Authentication by “what you have”

- Very common in the “physical world”, e.g., keys
- Digital world: Smart cards, USB tokens
- Private keys (e.g., for SSH)
- Can easily combine with password, e.g. on SSH private keys

Attacks and countermeasures

- **Stealing (or finding):** Protect possession
- **Copying:** Tamper-proof hardware, holograms, anti-counterfeiting techniques

Authentication by “what you have”

- Very common in the “physical world”, e.g., keys
- Digital world: Smart cards, USB tokens
- Private keys (e.g., for SSH)
- Can easily combine with password, e.g. on SSH private keys

Attacks and countermeasures

- **Stealing (or finding):** Protect possession
- **Copying:** Tamper-proof hardware, holograms, anti-counterfeiting techniques
- **Replay attack:** device-dependent, use challenge-response

Authentication by “what you are”

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html

Authentication by "what you are"

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html
- Retina scans

Authentication by “what you are”

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html
- Retina scans
- Voice match (distorted by cold, defeated by recordings)

Authentication by "what you are"

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html
- Retina scans
- Voice match (distorted by cold, defeated by recordings)
- Handwriting (low accuracy, easy to fake)

Authentication by "what you are"

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html
- Retina scans
- Voice match (distorted by cold, defeated by recordings)
- Handwriting (low accuracy, easy to fake)
- Keystroking, timing of keystrokes

Authentication by "what you are"

- Fingerprint (fake fingerprint, cut off finger)
  http://www.heise.de/video/artikel/iPhone-5s-Touch-ID-hack-in-detail-1966044.html
- Retina scans
- Voice match (distorted by cold, defeated by recordings)
- Handwriting (low accuracy, easy to fake)
- Keystroking, timing of keystrokes

When a password is compromised, change your password. What if your fingerprint is compromised?

Compromising fingerprints. . .

Politician’s fingerprint reproduced using photos of her hands

At a Chaos Computer Club convention, hacker Starbug suggests notable people wear gloves.

by **Megan Geuss** - Dec 30, 2014 2:05am CET

Share Tweet 111

Pluggable authentication modules

- Local login is not the only program that needs user authentication:
  - SSH (remote login)
  - Graphical login (GDM, LightDM)
  - Screen locks (screensaver)
  - su and sudo (more next lecture)

Pluggable authentication modules

- Local login is not the only program that needs user authentication:
  - SSH (remote login)
  - Graphical login (GDM, LightDM)
  - Screen locks (screensaver)
  - su and sudo (more next lecture)
- Idea: Centralize authentication, make functionality available through library
- This is handled by Pluggable Authentication Modules (PAM)

Pluggable authentication modules

- Local login is not the only program that needs user authentication:
  - SSH (remote login)
  - Graphical login (GDM, LightDM)
  - Screen locks (screensaver)
  - su and sudo (more next lecture)
- Idea: Centralize authentication, make functionality available through library
- This is handled by Pluggable Authentication Modules (PAM)
- Add a new module (e.g., for fingerprint authentication), directly available to all PAM enabled programs

PAM design

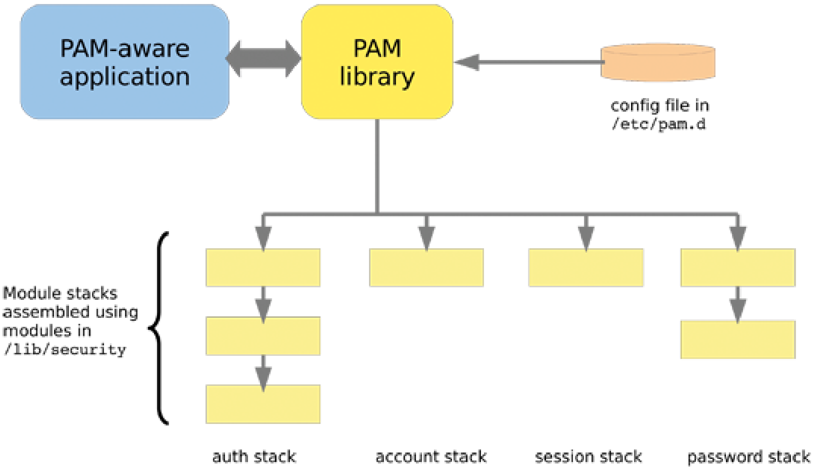


Image from http://www.tuxradar.com/content/how-pam-works

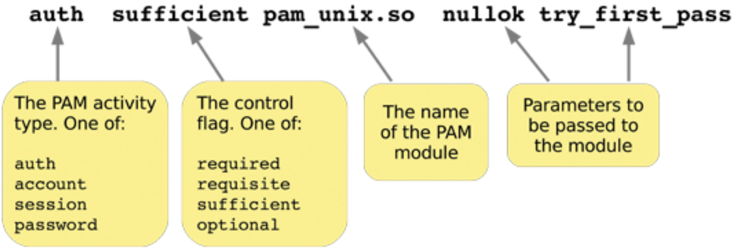
PAM activities

PAM knows 4 different authentication-related *activities*:

- **auth:** The activity of user authentication; typically by password, but can also use tokens, fingerprints etc.
- **account:** After a user is identified, decide whether he is allowed to log in. For example, can restrict login times.
- **session:** Allocates resources, for example mount home directory, set resource usage limits, print greeting message with information.
- **password:** Update the user's credentials (typically the password)

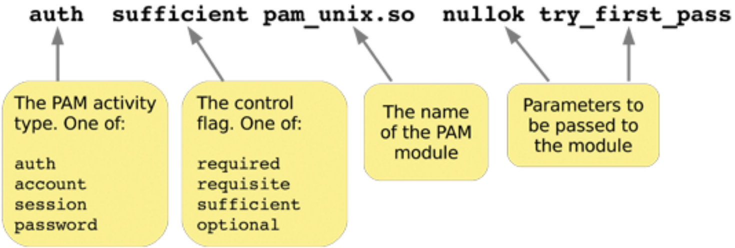
PAM configuration syntax

Configuration for program `progname` is in `/etc/pam.d/progname`



PAM configuration syntax

Configuration for program `progname` is in `/etc/pam.d/progname`



PAM control flags

- **requisite:** if module fails, immediately return failure and stop
- **required:** if module fails, return failure but continue
- **sufficient:** if module passes, return pass and stop
- **optional:** pass/fail result is ignored

Image source: http://www.tuxradar.com/content/how-pam-works

Examples of PAM modules

Name	Activities	Description
pam_unix	auth, session, password	Standard UNIX authentication through /etc/shadow passwords
pam_permit	auth, account, session, password	Always returns true
pam_deny	auth, account, session, password	Always returns false
pam_rootok	auth	Returns true iff you're root
pam_warn	auth, account, session, password	Write a log message to the system log
pam_cracklib	password	Perform checks of the password strength

Some PAM config examples

- Prevent all users from using su (/etc/pam.d/su)

```
auth        requisite  pam_deny.so
```

Some PAM config examples

- Prevent all users from using `su` (`/etc/pam.d/su`)

```
auth        requisite  pam_deny.so
```

- Prevent non-root users to halt (`/etc/pam.d/halt`)

```
auth        sufficient pam_rootok.so
auth        required   pam_deny.so
```

Some PAM config examples

- Prevent all users from using `su` (`/etc/pam.d/su`)

```
auth        requisite  pam_deny.so
```

- Prevent non-root users to halt (`/etc/pam.d/halt`)

```
auth        sufficient pam_rootok.so
auth        required   pam_deny.so
```

- Enforce passwords with at least 10 characters and at least 2 special characters, use SHA-512 for password hash (`/etc/pam.d/passwd`):

```
password    required   pam_cracklib.so minlen=10 ocredit=-2
password    required   pam_unix.so     sha512
```

Authentication over the network

- Large corporate networks want to keep user information central
- User is added to one central directory, can log into any machine

Authentication over the network

- Large corporate networks want to keep user information central
- User is added to one central directory, can log into any machine
- Various "simple" ways to set up the protocol:
  - Client sends password, server hashes and compares
  - Client sends hash, server compares
  - Server sends hash, client compares

Authentication over the network

- Large corporate networks want to keep user information central
- User is added to one central directory, can log into any machine
- Various “simple” ways to set up the protocol:
  - Client sends password, server hashes and compares
  - Client sends hash, server compares
  - Server sends hash, client compares
- Also more complex ways, e.g., challenge-response

Authentication over the network

- Large corporate networks want to keep user information central
- User is added to one central directory, can log into any machine
- Various “simple” ways to set up the protocol:
  - Client sends password, server hashes and compares
  - Client sends hash, server compares
  - Server sends hash, client compares
- Also more complex ways, e.g., challenge-response
- Possible disadvantage of central login server: single point of failure

Authentication over the network

- Large corporate networks want to keep user information central
- User is added to one central directory, can log into any machine
- Various "simple" ways to set up the protocol:
  - Client sends password, server hashes and compares
  - Client sends hash, server compares
  - Server sends hash, client compares
- Also more complex ways, e.g., challenge-response
- Possible disadvantage of central login server: single point of failure
- Different common protocols (NIS, LDAP, Kerberos)

NTLM and "pass the hash"

- Microsoft's LM and NTLM network authentication can send hash from the client, server compares hashes
- Attacker only needs to obtain the password *hash*
- The whole point of storing password hashes is gone
- Essentially, the hash becomes the password

NTLM and "pass the hash"

- Microsoft's LM and NTLM network authentication can send hash from the client, server compares hashes
- Attacker only needs to obtain the password *hash*
- The whole point of storing password hashes is gone
- Essentially, the hash becomes the password
- This attack is known as "pass the hash" attack

NTLM and "pass the hash"

- Microsoft's LM and NTLM network authentication can send hash from the client, server compares hashes
- Attacker only needs to obtain the password *hash*
- The whole point of storing password hashes is gone
- Essentially, the hash becomes the password
- This attack is known as "pass the hash" attack
- Conveniently automated in `metasploit`

NTLM and "pass the hash"

- Microsoft's LM and NTLM network authentication can send hash from the client, server compares hashes
- Attacker only needs to obtain the password *hash*
- The whole point of storing password hashes is gone
- Essentially, the hash becomes the password
- This attack is known as "pass the hash" attack
- Conveniently automated in `metasploit`
- Almost any larger Windows network still has NTLM somewhere

Part II

Authorization

Protection rings

- OS needs to control access to resources
- Idea: Access to resources only for highly-privileged code
- Non-privileged 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-privileged code
- Non-privileged 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)

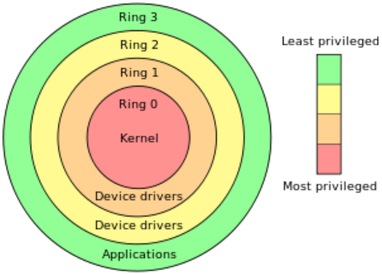


Image source: http://en.wikipedia.org/wiki/Protection_ring

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

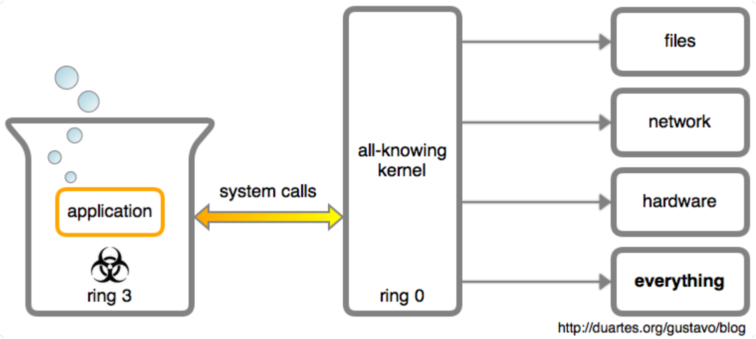
System calls and `strace`

- 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

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 $op : S \times O \to \mathcal{P}(X)$. 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)

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!

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

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*

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"

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 accidentally runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)
- This is a problem of all mainstream "classical" operating systems!

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 accidentally runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)
- This is a problem of all mainstream "classical" operating systems!
- 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 accidentally runs malware (more later in the lecture)
  - process operates on maliciously crafted input (in particular network processes)
- This is a problem of all mainstream "classical" operating systems!
- 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