50004 - Operating Systems - Lecture 16

Oliver Killane

24/12/21

Lecture Recording

Lecture recording is available here

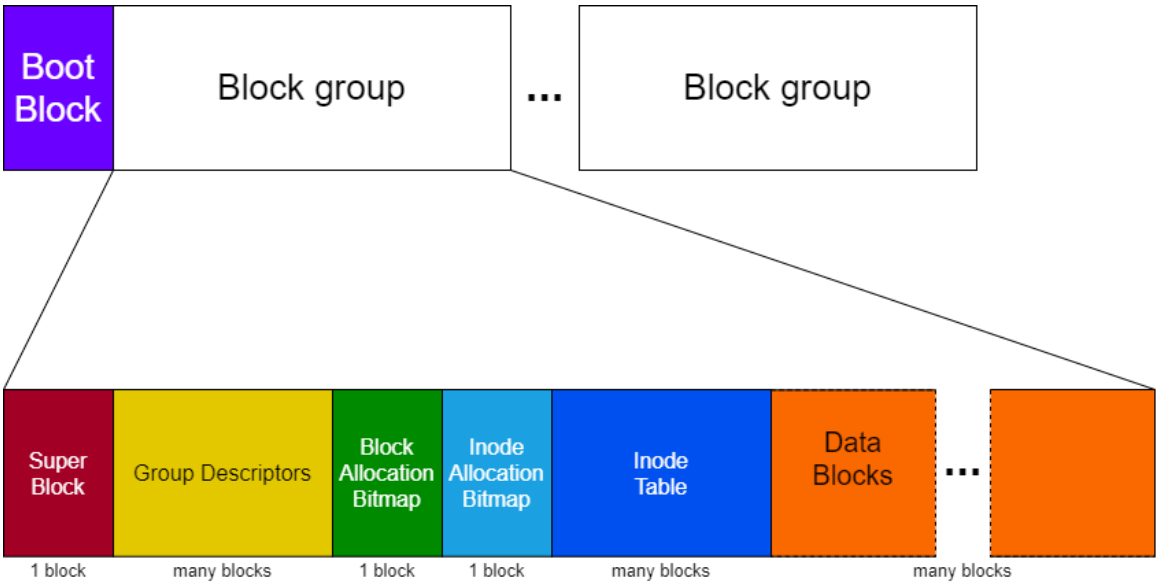
Linux ext2fs

The second **EXT**ended **F**ile **S**ystem is a high performance, robust file system (formerly the standard file system, now replaced by others but remains very portable).

- Typically uses block sizes of $1024, 2048, 4096, 8192$.
- 5% of blocks are reserved for the root (safety mechanism to ensure root processes can run even after a malicious process uses all available disk space).
- **ext2 inode** is used to represent files & directories. It stores access, permissions, and other metadata.
- **ext2 inode** uses 12 direct pointers, the 13^{th} is indirect, the 14^{th} doubly indirect and the 15^{th} triply indirect (allows fast access to small files, but allows for large files).

Block Groups

This filesystem makes use of **block groups** (clusters of contiguous blocks), and attempts to store related blocks (e.g from the same file) in the same block group to reduce seek time.



- **Super block** Critical information about the entire file system.
  - total number of blocks.
  - size of blocks, inode & block groups.
  - Metatdata for time accessed, mountings.

  Redundant copies are held in some block groups (reduces time to seek critical data).
- **Group Description** Contains pointers to the blocks containing:

  - Inode Allocation Bitmap
  - Block Allocation Bitmap
  - Inode Table
  - Other metatdata/accounting information.
- **Block Allocation Bitmap** Bitmap of blocks used/free within the data blocks section.
- **Inode Allocation Bitmap** Bitmap of entries containing an inode within the inode table.
- **Inode Table** Contains entries for every inode block in the group, spans over many blocks (many inodes, inode does not use up an entire block).
- **Data Blocks** Contains data blocks for files & directories.

In more detail ...

This excellent webpage goes into brilliant detail with example programs for **ext2fs**.

Security

Seurity Goals

The main objectives of security are:

- **Data Confidentiality** Preventing unauthorised read of data. e.g application secrets
- **Data Integrity** Preventing unauthorised write of data. e.g tampering with system clocks
- **System Availability** Preventing denial of service attacks. e.g monopolising a device (e.g hard drive)

Types of security in which we attempt to achieve said goals include:

- **People Security** Insider, social engineering (e.g blagging).
- **Hardware Security** e.g physical secureness of a system's hardware (e.g physically steal a disk to access data).
- **Software Security** Using flaws in software to access a system (e.g exploit a bug to run programs with higher permissions, such as superuser, without properly acquiring said permissions).

People Security

- **By Insiders** Need elevated privileges for their jobs, abuse these to compromise the security of the system.
- **Social Engineering** As people are often not security concious.

Phishing	Fake communications (e.g texts, emails) impersonating an organisation (can carry links to fake websites, or malicious attachments).
Blagging	More personal phishing (e.g ringing customer support).
Shouldering	Watching details (e.g passwords) in person.

  And many more.
- **Convenience** People ignore security requirements/sidestep them as it is easier (e.g repeated use of passwords, not properly verifying emails).
- **Ignorance** Lack of knowledge on security & incorrect assumptions (e.g email sender cannot be forged, internal emails are safe etc).

Hardware Security

- **Physical Access** Once physical access is acquired by the malicious party, they could:
  - Read & alter contents of attacked disks, memory.
  - Snoop on and forge network traffic (e.g if the machine is whitelisted for an internal organisation network).
  - Damage the machine to remove functionality (e.g commit arson against computer).
- **Hardware Flaws** Such as side channel attacks, badly implemented access control (Meltdown).

Side Channel Attacks

Gaining information using the implementation of a computer system. For example:

- **Cache Attack** Using cache accesses (e.g to a shared cache) by a victim to gain information.
- **Timing Attacks** Measuring time taken by a victim to gain information (e.g time taken to process a given key, compared with victim's unknown one).
- **Data Remnance** Reading data after the user has deleted it (e.g still in memory and has not be reset, files deleted on a drive are still present - blocks have just been marked as free).

Meltdown

Attack notes in **Lecture 10**. Modern CPUs reorder and speculatively execute instructions. By speculatively executing instructions that should fault (as they are accessing an invalid result) which change the state of cache based on the data accessed data can be improperly accessed.

1. CPU reorders instructions, speculatively executes access at **VICTIM_ADDR**.
2. If the data speculatively accessed is **0** then access **ASSAILANT_ADDR_F** else access **ASSAILANT_ADDR_T**.
3. The speculative executions is ignored (e.g if statement was **false**).
4. The assailant process now times its access to **ASSAILANT_ADDR_F** and **ASSAILANT_ADDR_T**. If the access is fast, the data is in cache, and hence we know the bit at **VICTIM_ADDR**.

Software Security

Compromising a system (unauthorized read, write & service denial) through software.

Common methods for exploits include:

- **Buffer Overflows** For example using **strcpy** on strings. If the string is larger than the buffer it may overflow, overwriting important program data. Use **strncpy** instead to limit bytes copied to the size of the buffer.
- **Integer Overflows** An example of this was a flaw in **SSH** you can read about here. By using integer overlow, **SSH** can accidentally make a hash table of size zero, effectively resulting in a buffer overflow when attempting to add data to the hashtable.

- **String Formatting** For example the **Log4J** vulnerability originating from string formatting in log messages allowing use of the **JNDI** java feature (which gets and locally executes java code). There is a great video explaining it here.

Access Control

- **Principals** Users, User groups or processes that may want to perform actions.
- **Authentication** Verifying the identity of users (**principals**). (e.g login system)
- **Authorisation** Only allow principals to perform actions when authorised. (e.g enforce read-/write/execute access on files)

Authentication

To authenticate the identity of a user we can use:

- **Personal Characteristics** Key based on user (usually biometrics).

  - Fingerprints.
  - Retinal scan.
  - Facial Recognition.
  - Signature & Signature motion analysis.
  - Typing rhythm analysis.

  Advantages include:

  - Characteristics hard to forge.
  - Convenience (e.g fingerprint and facial recognition becoming standard on smartphones).

  Disadvantages include:

  - Require special hardware (can be expensive).
  - Can have false positives/negatives (error associated with readers - e.g dirt on fingers, lighting conditions for facial recognition).
- **Possessions** Key is an item securely-kept by the user. This is the most widely used system.

  - RFID cards.
  - Implants.
  - Secure keys (e.g Yuibikey)
  - Can also consider devices such as smartphones, smartwatches which can be used as keys (e.g often for **2fa** - two factor authentication).

  Advantages include:

  - Convenience (rfid cards can be cheap to buy & replace).

  Disadvantages include:

  - If lost or stolen, can be used to impersonate.
  - Sometimes the cost of devices can be high (e.g complex locks, card scanners)
- **Knowledge** Key is a secret known to the user. (i.e passwords) Password turnover is important (longer a password is used, the greater time a cracker has to guess the password).

  A strong password (few english words, effectively random) can only be cracked by exhaustive search/brute force.

  Advantages include:

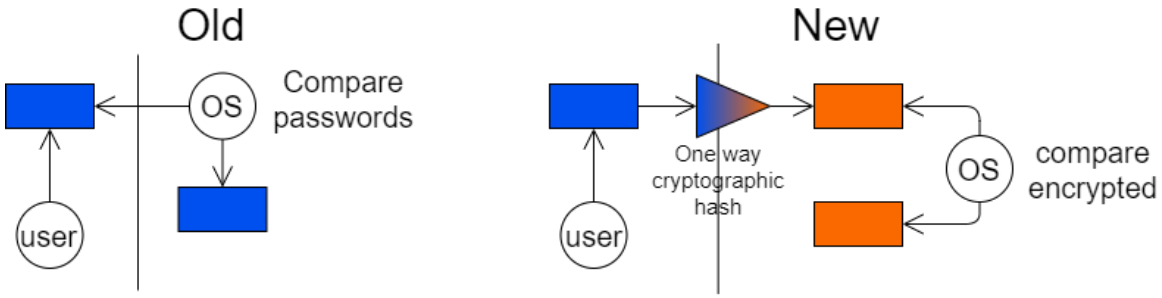
– Very cheap to implement (requires no special hardware).
– If password is kept secret, the system is secure.

Disadvantages include:

– Security diminished when passwords are reused.
– If passwords are stored, they are only as secure as the storage.
– Dictionary attacks be used for common passwords, or ones composed of words.
– If passwords are reused, the security of the systems using them are only as good as the weakest system using the password.

Password Verification

To verify passwords older systems store passwords in a protected file, newer systems store a one-way cryptographic hash of the password, and compare the hashed password entered with this. This ensures that even if the protected file is accessed, it is not possible to determine the password.



The older system is vulnerable to passwords being accidentally disclosed by a sysadmin, or if the sysadmin user is compromised.

The **etc/passwd** file used to contain plaintext passwords, currently holds user data.
The **etc/shadow** file contains hashed passwords.

One-Way Hash Function Password Protection

Can compute a hash easily, but not possible to invert the function.

$$Hash(Data) = Encrypted$$

The only feasable way of getting *Data* from *Encrypted* is brute force/extensive search of all data.

UNIX systems use hashing based on **DES** (Data Encryption Standard)

Rainbow Tables

Given a one-way hashing function, we can compute a table of hashes for all the most popular passwords.

This way given a hashed password, we can check if it is in the table. If it is, we know the password associated with the hash. We only need to compute the hash for each password once, and can continue to improve the table over time.

Password Salting

A method to prevent rainbow tables being effective. Another value s (the salt) is used.

The salt is created randomly upon account creation, and is stored with the hash (which takes the salt and password as parameters).

$$userid,\ salt,\ Hash(salt, password)$$

When the user logs in, the salt is used with the entered passsword to compute the hash.

$$\text{re-compute } Hash(salt \text{ (stored)}, password \text{ (entered)})$$

Hence computing a rainbow table for the hash in infeasible as for every password added to the table, every possible salt value must be hashed with it. This also ensures that two identical passwords are extremely unlikely to have the same hash.

Adobe Password Leak

In november 2013 $130,324,429$ passwords were leaked. As salting was not used, rainbow tables could be used to identify passwords.

Place	count	ciphertext	plaintext
1	1,911,938	EQ7fIpT7i/Q=	123456
2	446,162	j9p+HwtWWT86aMjgZFLzYg==	123456789
3	345,834	L8qbAD3jl3jioxG6CatHBw==	password
4	211,659	BB4e6X+b2xLioxG6CatHBw==	adobe123
5	201,580	j9p+HwtWWT/ioxG6CatHBw==	12345678
6	130,832	5djv7ZCI2ws=	qwerty
7	124,253	dQi0asWPYvQ=	1234567
8	113,884	7LqYzKVeq8I=	111111
9	83,411	PMDTbP0LZxu03SwrFUvYGA==	photoshop
10	82,694	e6MPXQ5G6a8=	123123

Authorisation

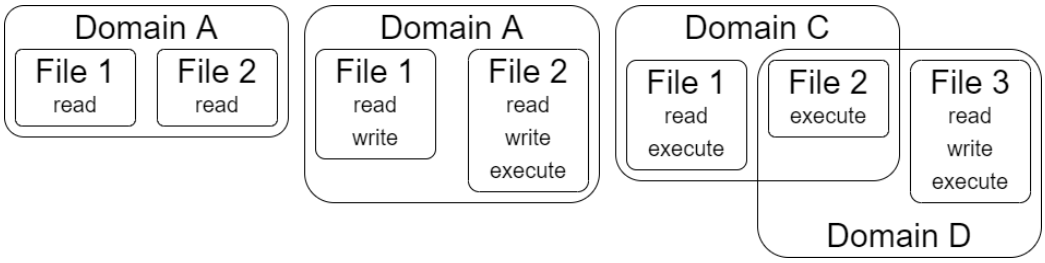
Determines who can access what objects (files, devices, directories, processes), and how they should access.

Principle of Least privilege

User should be given the minimum rights/privileges to carry out their assigned task.

This is not the case on many systems, to their detriment (as users can maliciously or accidentally misuse higher priveleges).

Access Rights are defined as a set of objects, with operations permitted on them. Each **principle** has a **domain** specifying the access rights.



An **Access control matrix** specifies the rights of each principal over all objects. An actual matrix is not used in reality, due to its enormous size over thousands of principals and millions of objects.

	Object 0	Object 1	Object 2	Object 3	Object 4
Principal 1	read		read & write	execute	
Principal 2		read & write		read & execute	
Principal 3	read & write	read	execute		read
Principal 4		read & execute			read, write & execute

Access Control Matrix Implementation

As a 2d array would be two large, there are two main implementations:

- **Access Control Lists** Each column of the matrix is stored as a list, and associated with each object.

  This list enforces which principals can access the object, and what operations they can perform.

- **capability List** Rows of matrix as lists associated with principals.

File Access on UNIX/Linux

- Users are the principals, and each has an associated user ID (UID). Each user can belong to one or more groups.
- The superuser (**root**) has UID 0 and all access rights (can access any file with all operations).
- Files are objects (e.g files, directories, sockets, pipes, block and character devices). Each file can belong to at most one group.

Access operations are read, write and execute:

Access Right	**File**	**Directory**
Read	Can read file contents.	Can list directory contents.
Write	Can write to file contents.	Can create/delete owned files.
Execute	Can execute the file (create as a new process).	Can enter directory & get access to the files.

Displaying Permissions

When listing a directory with the **-l** flag, the file permissions, owner, and other metadat is displayed. This is in **Lecture 14** under **file attributes**.

When a file is executed, it executes with the privileges of the user executing. Unless the **SUID** is set. In this case the file runs with the owner's privileges when executed (for example temporarily increasing privileges, but only for that program).

As a result each user effectively has 3 user IDs:

1. **Real UID** ID of the user that started the process
2. **Effective UID** Effective ID of the process (used for access control checks), when executing a file with the **SUID** flag this will be different to the real **UID**
3. **Saved UID** UID the effective UID can be switched to.

- When executing a file, the effective UID is $\begin{matrix}\text{user's real UID if not SUID}\\ \text{file owner's UID if SUID}\end{matrix}$.
- A non-root/superuser process may temporarily reduce its privileges, saving its effective UID as the saved UID so it can be restored later. It may do this to ensure **principle of least privelege**, and only hold elevated access rights when they are necessary.
- A non-root/superuser process can set their effective UID to their real UID or saved.