

CIA Triad

The **CIA Triad** is a foundational model in cybersecurity, representing three core principles that guide how we protect data and systems:

- Confidentiality: Prevents unauthorized access to sensitive information.
- Integrity: Ensures information remains accurate and unaltered.
- Availability: Ensures systems and data are accessible when needed by authorized users.



Confidentiality

The primary objective of *confidentiality* is to keep data secret from unauthorized users. It achieves this through a variety of techniques designed to prevent disclosure.

One of the most fundamental methods is **encryption**, which transforms plaintext into *ciphertext* using mathematical algorithms and keys.

The two main types of encryption are *symmetric* and *asymmetric*.

- Symmetric encryption: uses the same key for both encryption and decryption.
- Asymmetric encryption: uses a pair of public and private keys.

Asymmetric encryption is more secure for communication between untrusted parties but tends to be slower. Though regardless of the method, data should be encrypted *both in transit and at rest* to ensure protection during transfer and while stored.

Access control is another critical method of maintaining confidentiality. Access control restricts data and system access based on the identity of the user and the permissions assigned to that identity. This process consists of three core elements: identification, authentication, and authorization.

- Identification: when a user claims their identity, typically with a username.
- Authentication: requires the user to prove their identity through credentials such as a password, biometric scan, or token.
- Authorization: determines what actions the user is permitted to perform and what data they can access.

To strengthen access control, **multi-factor authentication** (MFA) is commonly implemented. MFA requires *two or more* independent forms of authentication:

- Something you know (e.g. password)
- Something you have (e.g. token, phone)
- Something you are (e.g. fingerprint, face, iris)

This significantly reduces the risk of unauthorized access even if one factor is compromised.

Another vital practice is the principle of **least privilege**, which ensures users and systems only have the minimum permissions necessary to perform their functions, reducing the attack surface.

Other techniques used to protect confidentiality include:

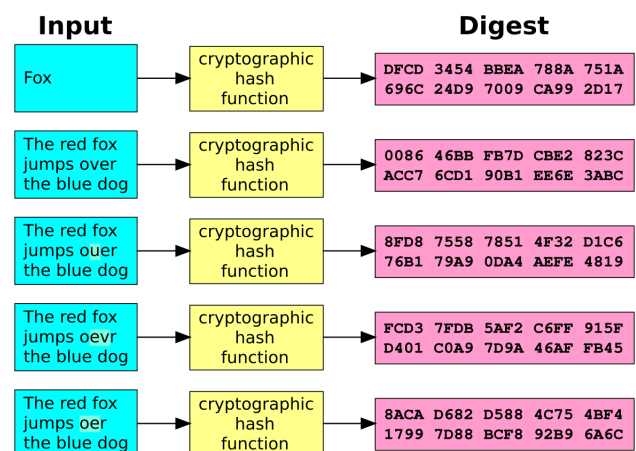
- Data classification: Labels data based on sensitivity (e.g. classified, secret)
- Obfuscation: Masks data elements
- Steganography: Hides data within other media (e.g. text hidden in images)

Together, these tools and principles work to ensure that sensitive information remains private and accessible only to those with a legitimate need.

Integrity

The primary goal of integrity is to ensure that data remains accurate, complete, and unaltered unless changed in a legitimate and authorized manner. This means protecting data from both intentional tampering and accidental modification, which can occur due to software bugs, human error, or malicious attacks.

One of the most common mechanisms for ensuring data integrity is the use of **cryptographic hashes**. A hash function takes an input—such as a file or string—and generates a *fixed-length alphanumeric string*, known as a hash value. Even the smallest change in the input will result in a completely different hash. This property, known as the *avalanche effect*, allows users or systems to verify whether data has been altered by comparing its current hash to a known-good reference.



For example, when downloading software, a user can verify the file's integrity by comparing its hash against the one provided by the software vendor. While hashes are excellent for detecting changes, they do not indicate what was changed or who changed it.

It's important to note the difference between cryptographic hashes and checksums, as both are used to verify integrity but serve different purposes and offer varying levels of protection.

- Checksums: Used primarily for accidental errors, not secure against tampering (e.g. CRC32)
- Hashes: Stronger, resistant to forgery and manipulation (e.g. SHA-256)

Digital signatures, on the other hand, provide both integrity and authenticity using asymmetric cryptography. A digital signature is created by first hashing the original data with a cryptographic hash function (e.g., SHA-256) and then encrypting that hash with the sender's private key. The recipient can then verify the signature by decrypting it with the sender's public key and comparing it to their own calculated hash of the received data. If the two values match, it proves not only that the data has not been altered (integrity), but also that it came from the legitimate source (authenticity). Digital signatures are widely used in secure systems including software updates, TLS/SSL certificates, and secure email protocols like PGP or S/MIME.

Beyond cryptographic methods, integrity can also be maintained through operational practices. **Version control** systems (e.g. Git) allow organizations to track and audit changes over time, helping to detect unauthorized modifications and roll back to a known-good state. Regular, validated **backups** ensure that data can be restored reliably if it becomes corrupted, lost, or compromised.

To further support data integrity, some systems implement **immutable logging**, which ensures that once a log entry is written, it cannot be altered or deleted. This provides a reliable audit trail essential for forensic analysis and compliance. Similarly, **configuration management** tools enforce consistency across environments by comparing current states to a defined baseline, alerting administrators to unauthorized or unintended changes.

By combining cryptographic techniques, lightweight integrity checks, and strong operational controls, organizations can build layered defenses that help detect, prevent, and recover from integrity violations across systems and data workflows.

Availability

The primary objective of availability is to ensure that systems, applications, and data are accessible to authorized users when needed. A system that lacks availability—even if it is

perfectly confidential and accurate—fails to meet its purpose. Achieving high availability requires a combination of planning, redundancy, fault tolerance, monitoring, and proactive maintenance.

To prevent service disruptions, organizations implement redundancy in various parts of their infrastructure.

- **Disk redundancy:** commonly achieved using RAID configurations, such as RAID 1 (mirroring) or RAID 5 (striping with parity), which allow for continued disk operations even after hardware failure.
- **Server redundancy:** organizations may deploy failover clusters or maintain standby servers that can take over automatically if the primary system fails.
- **Network redundancy:** often achieved through load balancing, which distributes traffic across multiple servers, or network interface card (NIC) teaming, which combines multiple physical connections for increased throughput and failover protection.
- **Power redundancy:** uninterruptible power supplies (UPS) and backup generators are used to keep systems running during electrical outages.

Keeping systems updated is also essential to availability. **Regular patching and maintenance** reduce the risk of unplanned downtime caused by software bugs or known vulnerabilities. Security patches in particular play a dual role by preventing exploits that could result in *denial-of-service* (DoS) conditions or broader system compromise.

To proactively detect and address issues that could impact availability, organizations use monitoring tools such as **Security Information and Event Management** (SIEM) systems and **Intrusion Detection Systems** (IDS). These tools provide visibility into system health and can generate alerts in response to suspicious activity or signs of failure.

Denial-of-service (DoS) attacks are a direct threat to availability. To defend against them, systems may implement **DoS protection** through rate limiting, blacklisting, geofencing, or through external services like content delivery networks (CDNs) that absorb and mitigate traffic spikes. Finally, capacity planning ensures that systems are equipped to handle future demand by forecasting usage patterns and scaling resources accordingly.

Scalability

Refers to the system's ability to grow or adapt to increased demand over time. While availability ensures that a system is functional and accessible, scalability ensures that the system can continue to function effectively under greater loads or during growth.

There are two main forms of scalability: vertical and horizontal.

- **Vertical scaling:** involves increasing the capacity of a single system—for example, by adding more memory, CPUs, or faster storage to a server. This can be effective for performance improvements but often reaches a limit based on hardware constraints.
- **Horizontal scaling:** involves adding more machines or instances to distribute the workload. This approach is generally more flexible and is often seen in distributed systems or load-balanced web services.

Scalability is especially important for businesses anticipating growth, seasonal demand fluctuations, or the need to support a large number of users.

Properly scaled systems avoid bottlenecks, reduce latency, and improve the user experience under high demand. It also contributes to reliability by allowing services to adapt rather than crash under pressure. However, scaling effectively requires thoughtful system architecture and careful planning, including load balancing, stateless application design, and distributed storage.

Elasticity

Elasticity builds upon the concept of scalability but introduces *automation*. While scalability often involves manual intervention to increase or decrease resources, elasticity enables systems to automatically allocate and deallocate resources in real time based on current demand. This is particularly relevant in cloud computing environments where resources such as virtual machines, storage, and processing power can be provisioned and released dynamically.

Elastic systems expand during peak usage periods and shrink during times of low activity, making them both cost-effective and performance-optimized. For example, an e-commerce platform might automatically increase the number of active web servers during a major sale event and reduce them afterward. This automatic responsiveness ensures that services remain available and responsive without wasting resources. Elasticity is typically managed through *infrastructure-as-code*, orchestration tools, and cloud-native features like AWS Auto Scaling or Azure Virtual Machine Scale Sets.

By enabling systems to respond immediately to workload changes, elasticity enhances both availability and performance. It minimizes the need for manual oversight while ensuring that systems remain resilient and performant under variable conditions.

Resilience

Refers to a system's ability to recover quickly from disruptions, faults, or failures while minimizing downtime and impact on users. Unlike high availability—which aims to prevent any downtime at all—resilience *accepts that failures* may occur but emphasizes recovery and continuity. A resilient system is designed with the expectation that components may fail, and includes mechanisms to handle, isolate, and recover from those failures with minimal interruption.

Resilient design often includes redundant infrastructure, automated failover, robust backup strategies, and fault-tolerant architectures.

For example, redundant storage ensures that data can still be accessed if a disk fails, and automated recovery scripts may restart failed processes or switch over to standby systems. Additionally, resilient systems make use of self-healing technologies that detect and respond to failures automatically, such as retrying failed requests or rebalancing workloads across healthy resources.

Regular data backups are a core element of resilience, as is having a tested disaster recovery plan. Resilience also includes maintaining immutable logs for auditing and forensics, which support post-incident analysis and accountability. From a design perspective, adopting a microservices architecture can enhance resilience by isolating failures within individual services, preventing them from cascading across the system.

Overall, resilience enables organizations to maintain trust and continuity, even in the face of hardware failures, cyberattacks, or natural disasters. It complements both availability and scalability by focusing not only on delivering services but on ensuring they can withstand and recover from unexpected conditions.

Practicals

Symmetric Encryption

As studied previously, symmetric encryption is a type of encryption where the same secret (or key) is used to encrypt and decrypt a designated piece of data. To see this in action, a Linux user may simply open the terminal on their machine and continue with the following procedure. Windows users may follow the same procedure through PowerShell using WSL ([click](#)).

```
user@kali: ~$ echo "SecretMessage" | openssl enc -aes-256-cbc -a -salt  
-pass pass:ultraSecretPassword  
  
U2FsdGVkX1/pp6mohwEANaW1tzAowPGG8YB3lPRkRnE=
```

The command inputted into the terminal above encrypts the plaintext string `SecretMessage` and outputs the ciphertext, in this case displayed as `U2FsdGVkX1/pp6mohwEANaW1tzAowPGG8YB3lPRkRnE=`.

The command features a lot of components to unpack, so here is a breakdown:

- `echo` -> common Linux command used to output text
- `"SecretMessage"` -> the plaintext input
- `|` -> pipes the output to the next command
- `openssl` -> a Linux-native cryptography toolkit
- `enc` -> specifies the encryption/decryption mode
- `-aes-256-cbc` -> uses AES with a 256-bit key in CBC mode
- `-a` -> outputs in Base64 (easier to read)
- `-salt` -> adds randomization to the output
- `-pass pass:ultraSecretPassword` -> the encryption key

```
user@kali: ~$ echo  
"U2FsdGVkX1/pp6mohwEANaW1tzAowPGG8YB3IPRkRnE=" | openssl enc  
-aes-256-cbc -a -d -salt -pass pass:ultraSecretPassword  
  
SecretMessage
```

To decrypt the message, the above command is executed. Here, the ciphertext calculated during the encryption process is passed as an argument and decrypted using the `-d` flag. It is important that the password and encryption method match what was used in the original encryption. For instance, if the user mismatched the encryption method or input the wrong password, it would provide the error as seen below.

```
user@kali: ~$ echo  
"U2FsdGVkX1/pp6mohwEANaW1tzAowPGG8YB3IPRkRnE=" | openssl enc  
-aes-128-cbc -a -d -salt -pass pass:ultraSecretPassword  
  
bad decrypt  
  
user@kali: ~$ echo  
"U2FsdGVkX1/pp6mohwEANaW1tzAowPGG8YB3IPRkRnE=" | openssl enc  
-aes-128-cbc -a -d -salt -pass pass:ultraWrongPassword  
  
bad decrypt
```


Asymmetric Encryption

As previously discussed, asymmetric encryption is a cryptographic method where two different keys are used: a public key to encrypt data and a private key to decrypt it.

The following practical demonstrates how asymmetric encryption works using OpenSSL in a Linux environment.

```
user@kali: ~$ openssl genpkey -algorithm RSA -out private.pem
user@kali: ~$ openssl rsa -pubout -in private.pem -out public.pem
```

The first command here generates a 2028-bit RSA private key and saves it as `private.pem` while the second extracts the corresponding public key and stores it in its own file named `public.pem`.

```
user@kali: ~$ cat private.pem

-----BEGIN PRIVATE KEY-----
MIIEvAIBADANBgkqhkiG9w0BAQEFAASCBywgg [.....]
-----END PRIVATE KEY-----

user@kali: ~$ cat public.pem

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A [.....]
-----END PUBLIC KEY-----
```

Though optional, at this point, the user can view either the public or private key. In the above screenshot, both keys are displayed in an abbreviated format, differentiated by their header and footer.

```
user@kali: ~$ echo "SecretMessage" > message.txt
user@kali: ~$ cat message.txt

SecretMessage
```

The above screenshot displays the creation of a text file, which is shown to contain the string `SecretMessage` in it, to be encrypted.

```
user@kali: ~$ openssl rsautl -encrypt -inkey public.pem -pubin -in
message.txt -out enc.bin
user@kali: ~$ cat enc.bin

?)??D?/qc??3|:Wi???2N?o?h?`lKmAOU?9?i>
?M?)?ä?^?Np??Btwa?Je???a+
```

At this point, the command used encrypts the text file using the public key previously generated.

- `-encrypt` -> tells OpenSSL to encrypt
- `-inkey public.pem -pubin` -> specifies the public key file
- `-in message.txt` -> plaintext input
- `-out enc.bin` -> encrypted binary format

As seen in the above screenshot, the encrypted file is unreadable, featuring various symbols and characters that seem random to the human eye.

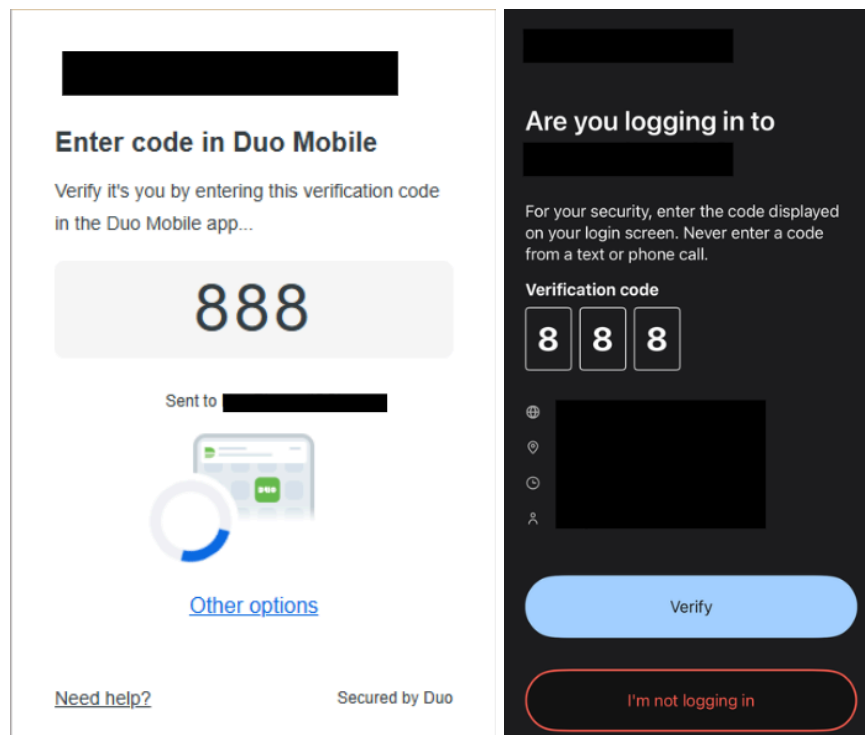
```
user@kali: ~$ openssl rsautl -decrypt -inkey private.pem -in enc.bin

SecretMessage
```


Finally, using the command in the above screenshot, the encrypted file is decrypted and displays the original plaintext.

MFA

As previously discussed Multi-Factor Authentication (MFA) may be accomplished by requiring a given user to authenticate using two different methods. For instance, two passwords is not much more secure than just one, as an attacker can often infiltrate and get access to both similarly. That being said, requiring a user to input a password and then input a code sent to their email or phone, or even authenticating through an app is much more secure, as it proves something the user knows and something the user has. Thus an attacker would not be able to get into a user account without knowing the password, which is easily stolen, and without having the device or email access, which is more difficult. There are not many practical examples to give besides simple demonstrations.



One of the most common forms of MFA today is in using an authenticator app. The example above is using Duo Mobile, though there are a number of more popular apps used for various websites and services. For instance, websites such as Discord and Bitwarden use Authy or Google Authenticator, while LinkedIn uses Microsoft Authenticator. In many cases, however, the user is given a choice of how to authenticate. For instance, Steam provides the user a choice between device verification through their app or a code sent to their phone or email. Many websites require MFA, others make it optional. An example of a code being sent via SMS can be seen below.



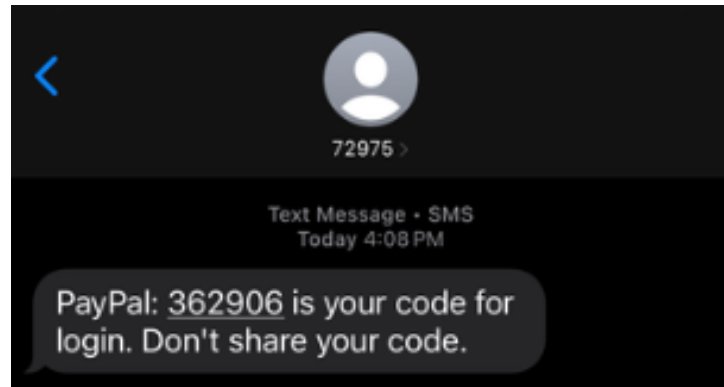
Enter the code we texted you

We sent it to [REDACTED]

[Send New Code](#)

[Submit](#)

[Choose Another Option](#)



There are many other authentication methods that are hardly indifferent but still to be acknowledged. As mentioned, Steam allows users to authenticate to their website using their Steam app; Google does this as well. Speaking of which, Google also allows security keys to authenticate a user. A security key is basically a physical device with a USB connection that authenticates a user when plugged into a computer; virtual security keys also work, often implemented with password managers such as BitWarden.

Hashes

Hashes are mathematical functions that transform input data into a fixed-length string — known as a hash value or digest. Even the smallest change in the input will result in a dramatically different hash. This property is known as the avalanche effect and is essential for verifying data integrity.

The following example demonstrates how a single change in file content affects the hash output.

```
user@kali: ~$ echo "Hello world" > file.txt
user@kali: ~$ sha256sum file.txt

1894a19c85ba153acbf743ac4e43fc004c891604b26f8c69e1e83ea2afc7c48f
```

The above screenshot displays the process to create a text file with the text `Hello world` in it, and calculate the hash value using SHA-256.

```
user@kali: ~$ echo "" > file.txt
user@kali: ~$ sha256sum file.txt

e16f1596201850fd4a63680b27f603cb64e67176159be3d8ed78a4403fdb1700
```

As seen in the above screenshot, the file was cleared of its previous contents and the hash value changed completely, demonstrating the sensitivity of hash functions and how they can be used.

Digital Signatures

Digital signatures are used to ensure integrity and authenticity of digital files and messages. They prove that the content has not been altered since signing (integrity), and that the content was signed by a specific identity (authenticity).

The following demo uses GPG, a popular open-source tool for digital signing and encryption, to demonstrate this.

```
user@kali: ~$ gpg --full-generate-key

Please select what kind of key you want? RSA and RSA
What keysize do you want? 2048
Key is valid for? 0
Real name: Bob Roberts
Email address: bob@roberts.com

pub  rsa2048 2025-05-28 [SC]
     9D7281CB86DE094B3AFC8296183170EB42C35741
uid                               Bob Roberts <bob@roberts.com>
sub  rsa2048 2025-05-28 [E]
```

The above screenshots displays the successful creation of a GPG key pair and associated metadata.

```
user@kali: ~$ echo "This is a confidential document." > document.txt
user@kali: ~$ gpg --output document.sig --detach-sign document.txt
user@kali: ~$ gpg --verify document.sig document.txt

gpg: Signature made Wed 28 May 2025 05:12:43 PM EDT
gpg:      using RSA key 9D7281CB86DE094B3AFC8296183170EB...
gpg: Good signature from "Bob Roberts <bob@roberts.com>" [ultimate]
```

In the above image, a simple plaintext file is made before being signed. The signature command can be broken down into two parts:

- `--output document.sig` -> output signature to a separate file
- `--detach-sign` -> keeps the original file separate from the signature

Lastly, the signature is verified. As seen in the screenshot, the terminal confirms the time and date the document was signed, what key was used, and who it came from.

Load Balancing

One common method of achieving network redundancy is load balancing. What this most practically does is exactly what the name suggests: it balances the load between multiple web servers. This keeps the website up and running well even when experiencing a lot of traffic.

In the following practical, two very basic web servers are made and Nginx is used to distribute incoming requests between them.

```
user@kali: ~$ sudo apt update
user@kali: ~$ sudo apt install docker.io -y
user@kali: ~$ sudo systemctl enable docker

Synchronizing state of docker.service with SysV service script with
/usr/lib/systemd/systemd-sysv-install.
Executing: /usr/lib/systemd/systemd-sysv-install enable docker

user@kali: ~$ sudo systemctl start docker
user@kali: ~$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

First things first, Docker must be installed, the process of which can be seen above. The installation is also verified with a test container.

```
user@kali: ~$ mkdir -p ~/.docker/cli-plugins
user@kali: ~$ curl -SL
https://github.com/docker/compose/releases/latest/download/docker-
compose-linux-x86_64 \
-o ~/.docker/cli-plugins/docker-compose
user@kali: ~$ docker compose version

Docker Compose version v2.36.2
```


Next, the Docker Compose binary has to be installed and placed in the appropriate directory, as seen in the above screenshot. The version does not matter; it is used here to confirm the installation was successful.

```
user@kali: ~$ mkdir load-balancer-lab
user@kali: ~$ cd load-balancer-lab
user@kali: ~$ mkdir web1 web2
user@kali: ~$ echo "<h1>This is Web Server 1</h1>" > web1/index.html
user@kali: ~$ echo "<h1>This is Web Server 2</h1>" > web2/index.html
```

A new directory is created for the exercise, along with two web server directories with static HTML pages.

```
user@kali: ~$ nano nginx.conf

events {}

http{
    upstream backend{
        server web1:80;
        server web2:80;
    }

    server{
        listen 80;

        location /{
            proxy_pass http://backend;
        }
    }
}
```

A Nginx configuration file is created here to tell Nginx to act as a reverse proxy that forwards traffic between Web Server 1 and Web Server 2. This is done in a round-robin manner, which essentially means it cycles through the two servers whenever there are multiple client requests. For instance, the first request will go to Web Server 1, the second will go to Web Server 2, the third will go to Web Server 1, and so on.

```
user@kali: ~$ nano docker-compose.yml
```

```
version: '3'

services:
  web1:
    image: nginx
    volumes:
      - ./web1:/usr/share/nginx/html:ro
    networks:
      - labnet

  web2:
    image: nginx
    volumes:
      - ./web2:/usr/share/nginx/html:ro
    networks:
      - labnet

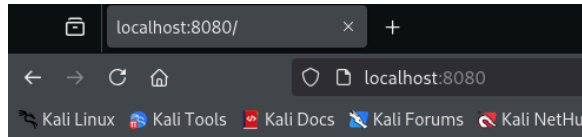
  loadbalancer:
    image: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "8080:80"
    depends_on:
      - web1
      - web2
    networks:
      - labnet

networks:
  labnet:
```

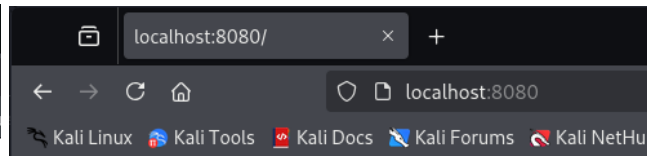
A second file is created with the above configuration. This configuration spins up three services: the two web servers and the Nginx load balancer which listens on the port 8080.

```
user@kali: ~$ sudo docker compose up --build
```

To finally launch the web services using Docker Compose, the above command is executed. Once the containers are up, open the machine's web browser and navigate to <http://localhost:8080>.



This is Web Server 1



This is Web Server 2

The message for Web Server 1 should appear upon opening the web page for the first time. Refreshing the page multiple times should result in the Web Server 2 message appearing. The messages should alternate as the page is refreshed, demonstrating the load balancing behavior.

Summary

Confidentiality

- Keep data secret from unauthorized users
- *Encryption*
 - Symmetric: One key for encryption and decryption.
 - Asymmetric: Public/private key pair for secure communication.
- *Access Control*
 - Identification: User claims identity (e.g., username).
 - Authentication: Proves identity (e.g., password, biometrics).
 - Authorization: Grants appropriate permissions.
- *Multi-Factor Authentication (MFA)* combines at least 2 of the following:
 - Something you know (password),
 - Something you have (token, phone),
 - Something you are (biometrics).
- *Principle of Least Privilege*: Limit access to the minimum necessary.
- Other techniques:
 - Data Classification: Labels based on sensitivity.
 - Obfuscation: Masks data.
 - Steganography: Hides data within other media.

Integrity

- Ensure data remains unaltered unless legitimately changed
- *Cryptographic hashes*
 - Unique output for any input (avalanche effect)
 - Detects any unauthorized changes
- *Checksums*
 - Used to detect accidental errors
 - Not as secure as hashes
- *Digital Signatures*
 - Combine hashing and asymmetric encryption
 - Provides integrity and authenticity of sender
- Operational practices:
 - Version Control (e.g., Git): Tracks changes, rollback support.
 - Validated Backups: Reliable recovery mechanism.
 - Immutable Logs: Unchangeable records for auditing.
 - Configuration Management: Detects unauthorized changes vs baseline.

Availability

- Keep systems/services accessible to authorized users
- *Redundancy*
 - Disk: RAID 1 (mirror), RAID 5 (parity)
 - Server: Failover clusters, standby systems
 - Network: Load balancing, NIC teaming
 - Power: UPS, backup generators
- *Maintenance and Monitoring*
 - Patch Management: Reduces vulnerability and downtime
 - Monitoring Tools:
 - **SIEM**: Aggregates and analyzes event data
 - **IDS**: Detect anomalies and threats
- *DoS Protection*
 - Rate limiting, blacklisting, geofencing.
 - CDNs: Absorb and distribute traffic loads.
 - Capacity Planning: Prepares for future demand.

Scalability

- Maintain performance under increased load
- Types:
 - Vertical Scaling: Add power to a single system (RAM, CPU).
 - Horizontal Scaling: Add more machines to distribute load.
- Benefits:
 - Reduces latency and bottlenecks.
 - Supports growth and reliability.
 - Requires thoughtful architecture (e.g., stateless design, distributed storage).

Elasticity

- Auto-adjust resources in real-time to match demand
- Scales up during high demand, down during low use.
- Reduces manual effort.
- Optimizes cost and performance.

Resilience

- Recover quickly from disruptions while minimizing user impact
- Design Principles:
 - Accept failure will happen—focus on recovery.
 - Redundancy: Backup hardware and paths.

- Failover Systems: Auto-switch during outages.
 - Self-Healing Mechanisms: Auto-restart, reroute, rebalance.
- Critical Tools:
 - Tested Backups & Disaster Recovery Plans
 - Immutable Logs: For audit and incident review.
 - Microservices Architecture: Isolates faults to prevent cascading failure.
- Outcome:
 - Maintains trust, continuity, and service delivery through disruptions.