# LPIC-3 Exam 303-300: Security

1st Edition

**Author:** ChatGPT

**Editor:** Omid Zeynalpour

# Contents:

# *Topic 331: Cryptography*

## 331.1 X.509 Certificates and Public Key Infrastructures (weight: 5)

- Understand X.509 certificates, X.509 certificate lifecycle, X.509 certificate fields and X.509v3 certificate extensions
- Understand trust chains and public key infrastructures, including certificate transparency
- Generate and manage public and private keys
- Create, operate, and secure a certification authority
- Request, sign, and manage server and client certificates
- Revoke certificates and certification authorities
- Basic feature knowledge of Let's Encrypt, ACME and certbot
- Basic feature knowledge of CFSSL

**X.509** is a standard for digital certificates used to verify the identity of a person, organization, or device. It defines the format of the certificate and the information that it contains.

**X.509 Certificate Lifecycle:**

The lifecycle of an X.509 certificate involves four stages:

- **Creation:** A certificate authority (CA) generates a public-private key pair for the entity requesting a certificate and creates a certificate signing request (CSR).
- **Validation:** The CA validates the identity of the entity requesting the certificate and verifies the information provided in the CSR.
- **Issuance:** The CA signs the certificate with its private key, creating a digital signature, and issues the certificate to the entity.
- **Expiration:** The certificate has a validity period after which it expires, and the entity must request a new certificate.

**X.509 Certificate Fields:**

An X.509 certificate contains the following fields:

- **Version:** Specifies the version of the X.509 standard used to create the certificate.
- **Serial number:** A unique identifier assigned by the CA to the certificate.
- **Subject:** The identity of the entity the certificate represents.
- **Issuer:** The identity of the CA that issued the certificate.
- **Validity period:** The start and end dates for which the certificate is valid.
- **Public key:** The public key of the entity to whom the certificate belongs.
- **Signature algorithm:** The algorithm used by the CA to sign the certificate.
- **Signature value:** The digital signature created by the CA using its private key.

**X.509v3 Certificate Extensions:**

X.509v3 introduces extensions to the standard certificate format, allowing for additional information to be included in the certificate. Some of the commonly used extensions include:

- **Key Usage:** Specifies the purpose of the public key, such as encryption or signature verification.
- **Subject Alternative Name:** Allows for additional subject identifiers to be included in the certificate, such as email addresses or IP addresses.
- **Basic Constraints:** Indicates whether the certificate is a CA certificate or an end-entity certificate.
- **Authority Key Identifier:** Provides information about the public key used to sign the certificate, allowing for chain building and revocation checking.
- **Extended Key Usage (EKU):** The EKU extension specifies the purposes for which the public key in the certificate can be used. The EKU extension allows a certificate issuer to specify the purposes for which a certificate can be used, beyond the basic purposes of authentication, digital signature, and encryption. For example, a certificate may be issued with an EKU indicating that it can be used only for email encryption, or only for code.

## Understand trust chains and public key infrastructures, including certificate transparency:

**Trust chains and Public Key Infrastructures (PKIs)** are used to establish trust in digital certificates and ensure the authenticity of the entities they represent. Certificate Transparency is a mechanism used to increase the transparency of PKIs.

A trust chain is a series of certificates issued by different Certificate Authorities (CAs) that links a particular digital certificate to a trusted root certificate. A digital certificate is considered valid if it is issued by a trusted CA and can be traced back to a trusted root certificate.

**A Public Key Infrastructure (PKI)** is a set of policies, procedures, and technologies used to create, manage, distribute, and revoke digital certificates. A PKI typically consists of one or more trusted CAs, a certificate management system, and software applications that use digital certificates for secure communication.

**Certificate Transparency** is a mechanism designed to provide an auditable record of all publicly trusted digital certificates. It achieves this by requiring CAs to publicly disclose all digital certificates they issue in a publicly accessible log. Certificate Transparency provides a means for detecting and mitigating certain types of certificate-based attacks, such as fraudulent certificates or mis-issuance by CAs.

In summary, trust chains and PKIs are used to establish trust in digital certificates and ensure their authenticity. Certificate Transparency is a mechanism used to increase the transparency of PKIs and provide an auditable record of all publicly trusted digital certificates.

## Generate and manage public and private keys:

Generating and managing public and private keys involves using cryptographic algorithms and software tools to create and store the keys securely. Here are the basic steps involved:

Choose the appropriate cryptographic algorithm: Common algorithms used for key generation include RSA, DSA, and ECDSA. The algorithm chosen should depend on the specific use case and security requirements.

Generate a public-private key pair: This involves using a software tool, such as OpenSSL or PuTTYgen, to create both the public and private keys. The private key should always be kept secret and stored securely, while the public key can be shared freely.

Protect the private key: The private key should be stored in a secure location, such as an encrypted USB drive or a hardware security module (HSM). It should be protected with a strong passphrase, and access to the key should be restricted to authorized personnel only.

Use the public key for encryption or digital signatures: The public key can be used to encrypt data or verify digital signatures. When encrypting data, the recipient uses the public key to decrypt the message. When verifying a digital signature, the public key is used to verify that the signature was generated by the corresponding private key.

Rotate keys regularly: For added security, it's recommended to generate new key pairs periodically, such as every six months or every year. This helps to ensure that any compromised keys are no longer in use and reduces the risk of data breaches.

Managing public and private keys involves ensuring their security, updating them regularly, and revoking any keys that have been compromised or are no longer in use. This can be done using a combination of security policies, processes, and software tools.

**Openssl commands:**

Here are some OpenSSL commands for generating and managing public and private keys:

Generate a private key:

```
openssl genpkey -algorithm RSA -out private_key.pem -aes256
```

This command generates a 2048-bit RSA private key and saves it in the "private_key.pem" file. The "-aes256" option encrypts the private key with a passphrase for added security.

Generate a public key from the private key:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

This command extracts the public key from the private key and saves it in the "public_key.pem" file.

Generate a CSR (Certificate Signing Request) for the public key:

```
openssl req -new -key private_key.pem -out csr.pem
```

This command generates a CSR for the public key and saves it in the "csr.pem" file. The CSR contains information about the organization that the certificate will be issued to, such as the domain name and contact information.

View the contents of a private key:

```
openssl pkey -in private_key.pem -text
```

This command displays the contents of the private key in human-readable format.

View the contents of a public key:

```
openssl rsa -in public_key.pem -pubin -text
```

This command displays the contents of the public key in a human-readable format.

Convert a private key to PKCS#8 format:

```
openssl pkcs8 -topk8 -in private_key.pem -out private_key.pkcs8.pem -nocrypt
```

This command converts the private key to PKCS#8 format, which is a widely used standard for private key storage. The converted key is saved in the "private_key.pkcs8.pem" file.

Convert a public key to DER format:

```
openssl rsa -in public_key.pem -pubin -outform DER -out public_key.der
```

This command converts the public key to DER format, which is a binary format that is sometimes used in web applications. The converted key is saved in the "public_key.der" file.

These are just a few examples of the many commands that can be used with OpenSSL for key generation and management. The specific commands and options used will depend on the specific use case and requirements.

## Create, operate, and secure a certification authority with OpenSSL commands:

Here are the basic steps:

Generate the CA private key and self-signed certificate:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout ca.key -out ca.crt -days 365
```

This command generates a new CA private key and self-signed certificate with a validity of 365 days. The private key is saved in "ca.key" and the certificate in "ca.crt".

Create a configuration file for the CA:

```
nano ca.cnf
```

Enter the following information in the configuration file:

```
[ ca ]
default_ca = my_ca

[ my_ca ]
new_certs_dir = <directory where new certificates should be saved>
certificate = <path to the ca.crt file>
private_key = <path to the ca.key file>
serial = <path to a file to hold the serial number>
database = <path to a file to hold the certificate database>
default_md = sha256
default_days = 365
policy = my_policy
```

Save and exit the configuration file.

Create a policy file for the CA:

```
nano ca.policy
```

Enter the following information in the policy file:

```
[ my_policy ]
commonName = supplied
```

Save and exit the policy file.

Create a serial number file for the CA:

```
echo '01' > ca.serial
```

Create a certificate database file for the CA:

```
touch ca.index
```

Create a CSR (Certificate Signing Request) for the server:

```
openssl req -new -newkey rsa:2048 -keyout server.key -out server.csr
```

This command generates a CSR for the server's certificate and saves the private key in "server.key" and the CSR in "server.csr".

Sign the server's CSR with the CA:

```
openssl ca -batch -config ca.cnf -policy my_policy -out server.crt -infiles server.csr
```

This command signs the server's CSR with the CA and creates a new certificate "server.crt".

Verify the server's certificate:

```
openssl verify -CAfile ca.crt server.crt
```

This command verifies the server's certificate against the CA's certificate to ensure it is valid.

These are the basic steps for creating and operating a CA with OpenSSL commands. There are additional steps that can be taken to further secure the CA, such as using a Hardware Security Module (HSM) to store the private key and limiting access to the CA's files and directories.

**Here are the basic steps for requesting, signing, and managing server and client certificates with OpenSSL commands:**

Create a Certificate Signing Request (CSR) for the server or client:

```
openssl req -new -newkey rsa:2048 -keyout server.key -out server.csr
```

This command generates a new private key and a CSR for the server. The private key is saved in "server.key" and the CSR in "server.csr". Replace "server" with "client" if you want to generate a CSR for a client certificate.

Submit the CSR to a Certificate Authority (CA) for signing:

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 365
```

This command signs the server's CSR with the CA's private key and creates a new certificate "server.crt". Replace "server" with "client" if you want to sign a client certificate. The "ca.crt" and "ca.key" files should be the certificate and private key of the CA, respectively.

Verify the server or client certificate:

```
openssl verify -CAfile ca.crt server.crt
```

This command verifies the server or client certificate against the CA's certificate to ensure it is valid.

Convert the server or client certificate and private key to a PKCS#12 format:

```
openssl pkcs12 -export -out server.p12 -inkey server.key -in server.crt
```

This command creates a PKCS#12 file "server.p12" containing the server or client certificate and private key. The file can be used for importing the certificate and key into various applications, such as web servers or email clients.

View the contents of a certificate:

```
openssl x509 -in server.crt -text
```

This command displays the contents of the certificate in a human-readable format.

Revoke a certificate:

```
openssl ca -revoke server.crt -config ca.cnf
```

This command revokes a certificate and updates the CA's certificate database. Replace "server.crt" with the name of the certificate to be revoked, and "ca.cnf" with the path to the CA's configuration file.

**Generate a Certificate Revocation List (CRL):**

```
openssl ca -gencrl -out crl.pem -config ca.cnf
```

This command generates a new CRL "crl.pem" that contains a list of revoked certificates. The file can be distributed to clients to check for revoked certificates.

These are just a few examples of the many commands that can be used with OpenSSL for requesting, signing, and managing server and client certificates. The specific commands and options used will depend on the specific use case and requirement

## Basic feature knowledge of Let's Encrypt, ACME, and Certbot:

**Let's Encrypt:**

- Free, automated, and open Certificate Authority (CA)
- Issues Domain Validated (DV) SSL/TLS certificates
- Goal is to make SSL/TLS encryption ubiquitous on the web

**ACME (Automated Certificate Management Environment):**

- Protocol used by Let's Encrypt to automate the issuance and renewal of SSL/TLS
- Certificates supports multiple challenges to verify domain ownership

**CertBot:**

- Open-source software tool that makes it easy to use Let's Encrypt SSL/TLS certificates on a server
- Automatically configures a web server to use Let's Encrypt certificates
- Can also automatically renew certificates before they expire
- Available for multiple operating systems (e.g. Linux, Windows, macOS)

Example use of Let's Encrypt with Certbot:

1. Install Certbot on a web server running Apache or Nginx
2. Run the command certbot --nginx or certbot --apache to obtain and install SSL/TLS certificate for the domain(s) hosted on the server
3. Certbot will communicate with Let's Encrypt's ACME server, validate domain ownership, and install the certificate on the web server
4. The web server is now configured to use HTTPS (HTTP over SSL/TLS) for secure communication

5. Certbot can also automatically renew the certificate before it expires to keep the website secure
6. The command certbot renew can be run regularly (e.g. as a cron job) to check for and renew expiring certificates.

Basic feature knowledge of **CFSSL (CloudFlare's SSL)**:

- Open-source toolkit for generating and managing SSL/TLS certificates
- Provides both a CLI and an HTTP API for certificate management
- Includes tools for certificate signing, certificate revocation, and managing certificate authorities (CA)
- Can be used to set up a local CA for an organization, or to sign certificates for internal services
- Written in Go and designed to be lightweight and fast
- Can be used in cloud, on-premise, or hybrid environments

Example use of **CFSSL**:

1. Install CFSSL on a server, or access the online API
2. Use the CFSSL CLI or API to generate a root CA certificate and private key
3. Use the root CA to sign intermediate CA certificates or server certificates as needed
4. Deploy the root CA certificate to all clients that need to trust the signed certificates
5. Install the signed server certificates on the appropriate servers
6. Configure the servers to use the signed certificate for SSL/TLS communication
7. Use CFSSL to revoke certificates as needed, and update the CRL (Certificate Revocation List) on all clients.

This example shows how CFSSL can be used to set up a secure certificate infrastructure for an organization, without relying on external certificate authorities like Let's Encrypt.

## 331.2 X.509 Certificates for Encryption, Signing and Authentication (weight: 4)

- Understand SSL, TLS, including protocol versions and ciphers
- Configure Apache HTTPD with mod_ssl to provide HTTPS service, including SNI and HSTS
- Configure Apache HTTPD with mod_ssl to serve certificate chains and adjust the cipher configuration (no cipher-specific knowledge)
- Configure Apache HTTPD with mod_ssl to authenticate users using certificates
- Configure Apache HTTPD with mod_ssl to provide OCSP stapling
- Use OpenSSL for SSL/TLS client and server tests

**SSL (Secure Sockets Layer) and TLS (Transport Layer Security)** are cryptographic protocols that provide secure communication over the internet. SSL was the predecessor of TLS, which is now the standard protocol for secure communication.

**Protocol Versions:**

- SSL 1.0 - never released publicly due to security vulnerabilities
- SSL 2.0 - also deprecated due to security vulnerabilities
- SSL 3.0 - still supported by some older systems but considered insecure and deprecated
- TLS 1.0 - widely used but has known vulnerabilities
- TLS 1.1 - improved security over TLS 1.0 but also has vulnerabilities
- TLS 1.2 - widely supported and considered secure
- TLS 1.3 - the latest version with significant security improvements over TLS 1.2

**Ciphers:**

Ciphers are the algorithms used to encrypt and decrypt data during SSL/TLS communication. Some commonly used ciphers are AES, DES, RC4, etc. The choice of cipher affects the security and performance of the SSL/TLS connection. Apache HTTPD with mod_ssl allows configuring ciphers by specifying a list of preferred ciphers in the configuration file.

Configuration of Apache HTTPD with mod_ssl:

1. To configure Apache HTTPD with mod_ssl to provide HTTPS service, install the **mod_ssl** module and generate or obtain an SSL/TLS certificate. The certificate should include the server's public key and be signed by a trusted certificate authority.
2. Enable the **mod_ssl** module in the Apache configuration file and configure the **SSLCertificateFile** and **SSLCertificateKeyFile** directives to point to the SSL/TLS certificate and its corresponding private key, respectively.
3. To support multiple domains on the same server with HTTPS, enable SNI (Server Name Indication) by setting the **SSLUseStapling** directive to "on".
4. To improve security, enable HSTS (HTTP Strict Transport Security) by setting the Header always set **Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"** directive. This sets a response header instructing web browsers to only access the website over HTTPS for a specified period of time.
5. To serve certificate chains, configure the **SSLCertificateChainFile** directive to point to the certificate chain file.
6. To authenticate users using certificates, enable client certificate authentication by configuring the **SSLVerifyClient** directive. This directive specifies the level of client certificate authentication required.
7. To provide OCSP (Online Certificate Status Protocol) stapling, enable the **SSLUseStapling** and **SSLStaplingCache** directives. OCSP stapling allows the server to provide the status of the SSL/TLS certificate without contacting the certificate authority.

**Using OpenSSL for SSL/TLS client and server tests:**

1. To test an SSL/TLS server using OpenSSL, use the s_client command. For example, to test an HTTPS server, run the command **openssl s_client -connect example.com:443**. This command will initiate a TLS handshake with the server and display information about the SSL/TLS connection.
2. To test an SSL/TLS client using OpenSSL, use the **s_server** command. For example, to test a client's SSL/TLS connection to a server, run the command **openssl s_server -cert server.crt -key server.key -accept 44330**. This command will start a server that listens for SSL/TLS connections on port 44330. Then initiate a connection from the client side using the s_client command as described above.

## 331.3 Encrypted File Systems (weight: 3)

- Understand block device and file system encryption
- Use dm-crypt with LUKS1 to encrypt block devices
- Use eCryptfs to encrypt file systems, including home directories and PAM integration
- Awareness of plain dm-crypt
- Awareness of LUKS2 features
- Conceptual understanding of Clevis for LUKS devices and Clevis PINs for TMP2 and Network Bound Disk Encryption (NBDE)/Tang

**Block device encryption and file system encryption** are two methods for securing data on a storage device. Block device encryption encrypts the entire storage device, while file system encryption encrypts specific files or directories.

**Block device encryption** involves encrypting the entire block device, which includes the file system and all its contents. This means that all data on the device is encrypted, including the file system structure itself. One way to encrypt block devices on Linux systems is to use dm-crypt with LUKS1.

**LUKS1 (Linux Unified Key Setup) is a disk encryption** specification that provides a standard format for encrypted disk volumes. It is based on dm-crypt, which is a Linux kernel-level encryption module. To use dm-crypt with LUKS1, you need to create a LUKS partition on the block device, set up the encryption parameters and key, and then mount the partition using the decrypted data.

**here's an example of using dm-crypt with LUKS1 to encrypt a block device:**

First, make sure that the **cryptsetup** package is installed on your Linux system.

Identify the block device that you want to encrypt. For example, if you want to encrypt the partition **/dev/sdb1**, you would run the command **lsblk** to list all the available block devices, and then locate the partition you want to encrypt.

Create a LUKS partition on the block device by running the following command:

```
sudo cryptsetup luksFormat /dev/sdb1
```

This will prompt you to create a passphrase for the partition. Make sure to choose a strong passphrase and remember it, as you will need it to unlock the partition later.

Open the encrypted partition by running the following command:

```
sudo cryptsetup open /dev/sdb1 encrypted_partition
```

This will prompt you to enter the passphrase you created in the previous step. Once you enter the correct passphrase, the partition will be unlocked and accessible at the device mapper path **/dev/mapper/encrypted_partition**.

Create a file system on the encrypted partition, using the device mapper path you obtained in the previous step:

```
sudo mkfs.ext4 /dev/mapper/encrypted_partition
```

Mount the encrypted partition as a regular file system, using the device mapper path and a mount point of your choice:

```
sudo mount /dev/mapper/encrypted_partition /mnt/encrypted
```

Now, any files that you store in the /mnt/encrypted directory will be automatically encrypted and decrypted on-the-fly using the LUKS encryption parameters and key that you set up earlier.

**File system encryption,** on the other hand, involves encrypting specific files or directories. One way to encrypt file systems on Linux systems is to use **eCryptfs**.

**eCryptfs** is a stacked cryptographic file system that provides on-the-fly encryption and decryption of individual files or directories. It is similar to dm-crypt with LUKS1 in that it uses Linux kernel-level encryption, but it is designed for file-level encryption rather than block-level encryption.

To use eCryptfs, you need to create an encrypted directory or mount point, which can be used to store encrypted files. You can then mount the encrypted directory or mount point using the ecryptfs mount utility. One advantage of eCryptfs is that it allows for per-file encryption, so you can encrypt only the files that need to be protected, rather than the entire file system.

**PAM (Pluggable Authentication Modules)** integration can be used to authenticate users and provide access to encrypted files or directories. This allows users to enter their credentials once, and then access encrypted data without having to enter their password again. PAM integration is available for both dm-crypt with LUKS1 and eCryptfs.

**dm-crypt** is a disk encryption subsystem in Linux that provides transparent encryption for block devices. It operates at the device mapper level and can be used with various encryption algorithms such as AES, Blowfish, Serpent, Twofish, etc.

**Plain dm-crypt** refers to the basic implementation of dm-crypt without any header or key management infrastructure. This means that there is no standard method for storing encryption keys or metadata, so it requires manual setup and administration.

Plain dm-crypt can be useful for simple use cases where disk encryption is needed, but it lacks the convenience and security features of more advanced disk encryption solutions like LUKS.

To use plain dm-crypt, you can use the cryptsetup tool to create a mapped device, specifying the encryption algorithm and key in the command line. However, be aware that this approach is less secure and convenient than using LUKS.

**eCryptfs:**

- An open-source, stackable, cryptographic file system for Linux
- Provides transparent encryption for individual files or entire directories
- Uses the Linux kernel and a FUSE (Filesystem in Userspace) module to provide encryption
- Integrates with the Linux PAM (Pluggable Authentication Modules) system to provide user-level encryption

**ecryptfs-* commands:** Command-line tools for managing and using eCryptfs file systems

Includes tools for creating and managing encrypted directories, configuring key management, and mounting/unmounting encrypted file systems

**mount.ecryptfs, umount.ecryptfs:**

The standard Linux commands for mounting and unmounting eCryptfs file systems

Provide options for specifying the source and target directories, key management options, and other mount options

**pam_ecryptfs:**

- A PAM (Pluggable Authentication Modules) module for Linux that integrates with eCryptfs
- Allows users to have encrypted home directories, automatically mounted and unlocked when the user logs in
- Integrates with the Linux login system to provide seamless user-level encryption

In summary, eCryptfs provides transparent encryption for individual files and directories on Linux, and the ecryptfs-* commands and mount.ecryptfs/umount.ecryptfs provide a way to manage and use eCryptfs file systems. The pam_ecryptfs module integrates eCryptfs with the Linux PAM system to provide user-level encryption.

**Example of ecryptfs commands:**

Create an encrypted directory:

```
$ mkdir ~/private
```

```
$ chmod 700 ~/private
```

```
$ ecryptfs-setup-private ~/private
```

Mount an encrypted directory:

```
$ mount.ecryptfs ~/private ~/private
```

Unmount an encrypted directory:

```
$ umount ~/private
```

Add a key for a new user to access an encrypted directory:

```
$ ecryptfs-add-passphrase --fnek
```

```
$ ecryptfs-manager add [new user name] [mount point] [fnek signature]
```

Remove a key for a user to access an encrypted directory:

```
$ ecryptfs-manager remove [user name] [mount point]
```

These are basic examples of how the ecryptfs-* commands can be used to manage and use eCryptfs file systems. The exact commands and options will depend on the specific use case and the configuration of the system.

**Conceptual understanding of Clevis for LUKS devices and Clevis PINs for TMP2 and Network Bound Disk Encryption (NBDE)/Tang**

**Clevis:**

- An open-source tool for automating the unlocking of LUKS-encrypted devices at boot time
- Uses "tang" to securely store encryption keys, and "clevis" to automate the unlocking process
- Integrates with other key management systems (e.g. TPM, PKCS#11, Hashicorp Vault) to provide secure key storage

**Clevis PINs:**

- A way to securely store encryption keys in a TPM (Trusted Platform Module) or on a network-bound key server
- The encryption key is encrypted with a key derived from a user-entered PIN, and the encrypted key is stored in the TPM or on the key server
- The Clevis tool can use the encrypted key to unlock the LUKS-encrypted device at boot time, without requiring the user to manually enter the encryption key

**TPM2:**

- Trusted Platform Module version 2, a secure hardware component found in some computers
- Can store encryption keys and perform cryptographic operations, providing secure key storage
- Can be used with Clevis to store encrypted encryption keys for LUKS-encrypted devices

**Network Bound Disk Encryption (NBDE)/Tang:**

- A method for securely storing encryption keys on a network-bound key server

- Uses a tool called "tang" to store the encrypted encryption keys, and a tool called "clevis" to automate the unlocking of LUKS-encrypted devices
- Provides secure key storage, and allows central management of encryption keys for multiple servers.

In summary, Clevis and Clevis PINs provide a way to securely store encryption keys for LUKS-encrypted devices, and automate the unlocking process at boot time. This helps to ensure that the encrypted data remains secure, even if the device is lost or stolen.

## 331.4 DNS and Cryptography (weight: 5)

- Understand the concepts of DNS, zones and resource records
- Understand DNSSEC, including key signing keys, zone signing keys and relevant DNS records such as DS, DNSKEY, RRSIG, NSEC, NSEC3 and NSEC3PARAM
- Configure and troubleshoot BIND as an authoritative name server serving DNSSEC secured zones
- Manage DNSSEC signed zones, including key generation, key rollover and re-signing of zones
- Configure BIND as an recursive name server that performs DNSSEC validation on behalf of its clients
- Understand CAA and DANE, including relevant DNS records such as CAA and TLSA
- Use CAA and DANE to publish X.509 certificate and certificate authority information in DNS
- Use TSIG for secure communication with BIND
- Awareness of DNS over TLS and DNS over HTTPS
- Awareness of Multicast DNS

### Understand the concepts of DNS, zones and resource records

DNS (Domain Name System) is a hierarchical system that translates human-readable domain names (such as "example.com") into IP addresses (such as "93.184.216.34") that computers can understand. DNS uses a distributed database system to store information about domain names and their associated IP addresses, allowing computers to quickly look up the IP address of a domain name.

DNS is organized into a hierarchy of zones, with the root zone at the top, followed by top-level domains (such as .com, .org, .net, etc.), second-level domains (such as example.com), and so on. Each zone is managed by a domain name registrar, and can contain one or more resource records.

A resource record is a type of data record in a DNS database that provides information about a particular domain name or IP address. Some common types of resource records include:

- **A (Address) record:** maps a domain name to an IP address
- **CNAME (Canonical Name) record:** maps an alias (such as "www") to the canonical name of a domain (such as "example.com")
- **MX (Mail Exchange) record:** specifies the mail server responsible for handling email for a domain
- **NS (Name Server) record:** specifies the authoritative name servers for a domain
- **SOA (Start of Authority) record:** provides administrative information about a zone, such as the name of the zone's primary name server and the email address of the zone administrator

**Resource records** are stored in zone files, which are plain text files that contain information about a particular DNS zone. The format of a zone file is standardized and includes information such as the domain name, the authoritative name servers for the zone, and the resource records for the zone.

When a DNS lookup is performed, the client sends a query to a DNS server, asking for the IP address of a particular domain name. The DNS server then searches its database for the appropriate resource record and returns the IP address to the client. If the DNS server does not have the requested record in its database, it will forward the query to another DNS server higher up in the hierarchy until the record is found.

**DNSSEC (Domain Name System Security Extensions)** is a security extension to the DNS that provides authenticity and integrity to DNS data. It uses digital signatures and public-key cryptography to secure DNS data and protect against tampering.

DNSSEC protects against DNS spoofing, which is a type of cyber attack where a malicious attacker modifies DNS responses to redirect users to fake or malicious websites. With DNSSEC, the DNS data is signed with a digital signature that can be verified by resolvers, ensuring that the data has not been tampered with.

To manage DNSSEC signed zones, the following steps are involved:

**Key Generation:**

Generate a key pair (public and private key) for the zone using a key generation tool like **dnssec-keygen**

Publish the public key in the DNS as a DNSKEY resource record.

**Key Signing:**

Sign the zone data with the private key.

Publish the signed zone data, including the RRSIG (resource record signature) records, in the DNS.

**Key Rollover:**

- Generate a new key pair for the zone.
- Publish the new public key in the DNS as a DNSKEY resource record.
- Sign the zone data with the new private key.
- Publish the signed zone data, including the new RRSIG records, in the DNS.
- Remove the old public key from the DNS.

**Re-signing of Zones:**

Periodically re-sign the zone data to refresh the signatures and keep them valid.

Here is a simple diagram that represents the basic steps in managing DNSSEC signed zones:

**Examples of DNSSEC**

1. A user tries to access a secure website, for example, www.example.com.
2. The user's computer sends a DNS query to a resolver for the IP address of www.example.com.
3. The resolver retrieves the DNS data for www.example.com from the authoritative DNS server.
4. The authoritative DNS server returns the IP address of www.example.com and the DNSSEC signatures for the data.
5. The resolver checks the DNSSEC signatures to verify that the DNS data has not been tampered with.
6. The DNSSEC signatures are based on a chain of trust that starts with a root zone, which contains the public keys of the top-level domains (TLDs).
7. The resolver checks each signature in the chain of trust, starting from the TLD to the specific domain (www.example.com) being queried, to ensure that the signatures are valid and that the DNS data is authentic.
8. If the signatures are valid, the resolver returns the IP address of www.example.com to the user's computer.
9. The user's computer can then establish a secure connection to the website and access its content.

In this example, we can see how DNSSEC provides a secure way for users to access websites by verifying the authenticity and integrity of the DNS data. The DNSSEC signatures protect against DNS spoofing, where a malicious attacker modifies DNS responses to redirect users to fake or malicious websites. By using DNSSEC, users can be confident that the website they are accessing is the real one and that their data is protected.

**what are key signing keys, and zone signing keys?**

Key signing keys (KSKs) and zone signing keys (ZSKs) are two types of keys used in DNSSEC to secure DNS data.

**Key Signing Keys (KSKs):** Key Signing Keys are long-lived public/private key pairs used to sign the DNSSEC trust anchor, which is the root of the chain of trust. KSKs are used to sign the DNSKEY records, which are the public keys of the TLDs, and are published in the root zone. The KSKs are stored offline and are used infrequently to sign the DNSKEY records.

**Zone Signing Keys (ZSKs):** Zone Signing Keys are short-lived public/private key pairs used to sign the actual DNS data of a zone, such as A, MX, or NS records. ZSKs are used to sign the resource records for a specific domain, such as www.example.com. The ZSKs are published in the zone data, along with the signed resource records, and are used frequently to sign the zone data.

In summary, the KSKs are used to sign the DNSKEY records, which form the chain of trust in DNSSEC, while the ZSKs are used to sign the actual DNS data of a zone. By using both KSKs and ZSKs, DNSSEC provides a secure way to verify the authenticity and integrity of DNS data and protect against DNS spoofing.

**Configure the DNS server to support DNSSEC:**

The DNS server must support DNSSEC and be configured to use the KSK and ZSK to sign and verify the DNS data.

The DNS server must also be configured to serve the signed DNSKEY records and the signed zone data for the domain.

**DNS records such as DS, DNSKEY, RRSIG, NSEC, NSEC3 with examples:**

DNSSEC uses several types of DNS records to secure the DNS data:

**DS (Delegation Signer) record:** A DS record is a resource record that is stored in a parent zone and is used to delegate a subzone to a child zone. The DS record contains the hash of a DNSKEY record for the child zone, and is used to verify the authenticity of the child zone's public keys.

Example:

**example.com.      IN      DS      12345 7 1 B33F1337A...**

**DNSKEY record:** A DNSKEY record is a resource record that contains the public key for a TLD or a child zone. The DNSKEY record is signed by the Key Signing Key (KSK) and is used by resolvers to verify the authenticity of the public keys for a TLD or child zone.

Example:

**example.com.      IN      DNSKEY  256 3 7 ( AwEAAc... )**

**RRSIG (Resource Record Signature) record:** An RRSIG record is a resource record that contains a digital signature for a specific type of resource record. The RRSIG record is used to verify the authenticity and integrity of the signed resource record.

Example:

**example.com.      IN      RRSIG   A 7 2 3600 ( ... )**

**NSEC (Next Secure) record:** An NSEC record is a resource record that is used to prove the non-existence of a resource record. The NSEC record lists the next existing resource record in the zone and the types of resource records that exist for that name

example.com.      IN      NSEC    example.com. A RRSIG NSEC

**NSEC3 (Next Secure version 3) record:** An NSEC3 record is a resource record that is used to prove the non-existence of a resource record. The NSEC3 record is similar to the NSEC record, but uses a hash function to obscure the names of the resource records in the zone.

Example:

**example.com.      IN      NSEC3   1 0 10 ( B33F1337A... )**

These records are used in combination to secure the DNS data and ensure that clients can access the correct websites and services. The exact details of how these records are used will depend on the specifics of the DNSSEC implementation and the DNS server software.

**Understand CAA and DANE, including relevant DNS records such as CAA and TLSA:**

CAA (Certification Authority Authorization) and DANE (DNS-Based Authentication of Named Entities) are two methods for securing domain names and ensuring that only authorized Certificate Authorities (CAs) can issue SSL/TLS certificates for a domain.

**CAA** is a DNS resource record that specifies which Certificate Authorities (CAs) are authorized to issue SSL/TLS certificates for a domain. CAA records are used by CAs to check if they are authorized to issue certificates for a particular domain. The record can contain multiple flags that specify which actions are permitted, such as "issue" or "issuewild."

**DANE** is a method of specifying the location of a trusted SSL/TLS certificate using a TLSA record in the domain's DNS. The TLSA record contains information about the certificate, such as its hash value and usage type, and allows domain owners to specify the certificate that should be trusted for secure communication. DANE helps to mitigate the risk of Man-in-the-Middle (MitM) attacks and to ensure that SSL/TLS certificates are only issued by trusted CAs.

**Both CAA and DANE** records are used to improve the security of SSL/TLS certificates and to ensure that only authorized CAs are able to issue certificates for a particular domain. CAA records specify which CAs are authorized, while DANE records specify the certificate that should be trusted for secure communication.

**Here's an example of how to use CAA records:**

Decide which Certificate Authorities (CAs) are authorized to issue certificates for your domain.

Create a CAA record in your domain's DNS zone file using the following syntax:

**example.com. IN CAA 0 issue "ca.example.com"**

This CAA record specifies that only the "ca.example.com" CA is authorized to issue certificates for "example.com." The "0" flag specifies that all other actions are prohibited.

**Here's an example of how to use DANE records:**

Obtain the SSL/TTL certificate you want to use for your domain.

Generate the TLSA record using the following syntax:

**_443._tcp.example.com. IN TLSA 3 1 1 ( certificate hash value )**

This TLSA record specifies that the SSL/TLS certificate with the specified hash value should be trusted for secure communication on port 443 for the "example.com" domain. The "3" flag specifies that the certificate is stored in the DANE-EE(3) format, "1" specifies that the full certificate is used, and "1" specifies that a SHA-256 hash is used.

In this example, the CAA record allows domain owners to specify which CA is authorized to issue certificates for their domain, while the TLSA record allows domain owners to specify the certificate that should be trusted for secure communication. By using both CAA and DANE, domain owners can have greater control over their certificate and certificate authority information, and mitigate the risk of MitM attacks.

**Use TSIG for secure communication with BIND:**

TSIG (Transaction SIGnature) is a method used to secure communication between DNS servers. Here's an example of how to use TSIG with BIND:

Generate a shared secret key using the dnssec-keygen utility:

**dnssec-keygen -a HMAC-SHA256 -b 256 -n HOST server1.example.com**

**This will generate two files: a private key file (Kserver1.example.com.+165+XXXXX.private) and a key file (Kserver1.example.com.+165+XXXXX.key).**

Configure **the primary (master) DNS server** with the key:

```
key "server1.example.com" {
   algorithm HMAC-SHA256;
   secret "secret key value from the .private file";
};
```

Configure the secondary (slave) DNS server with the key:

```
key "server1.example.com" {
   algorithm HMAC-SHA256;
   secret "secret key value from the .private file";
};

server 192.168.1.1 {
  keys {
   "server1.example.com";
  };
};
```

Update the zone transfer (AXFR) definition on the primary server to use the key:

```
zone "example.com" {
   type master;
   file "db.example.com";
   allow-transfer { key "server1.example.com"; };
};
```

Update the zone transfer (AXFR) definition on the secondary server to use the key:

```
zone "example.com" {
   type slave;
   file "db.example.com";
   masters { 192.168.1.1 key "server1.example.com"; };
};
```

In this example, the shared secret key is used to authenticate the communication between the primary and secondary DNS servers. This helps to ensure that the data transferred between the servers is not tampered with during transmission.

**Awareness of DNS over TLS and DNS over HTTPS:**

DNS over TLS (DoT) and DNS over HTTPS (DoH) are two security protocols for transmitting Domain Name System (DNS) queries and responses over an encrypted connection, providing enhanced privacy and security compared to traditional DNS.

DNS over TLS uses the Transport Layer Security (TLS) protocol to encrypt DNS queries and responses, while DNS over HTTPS uses the Hypertext Transfer Protocol Secure (HTTPS) to perform the same task. Both protocols aim to prevent eavesdropping and tampering of DNS traffic by encrypting the data being transmitted between the client and the DNS resolver.

**Awareness of Multicast DNS**

Multicast Domain Name System (mDNS) is another type of DNS resolution protocol, which uses multicast Domain Name System (DNS) packets to resolve hostnames to IP addresses within small networks that do not have a conventional DNS server. mDNS is used for local network service discovery, and it allows devices to publish and discover services and hostnames on a local network without the need for a centralized DNS server.

# *Topic 332: Host Security*

## 332.1 Host Hardening (weight: 5)

- Configure BIOS and boot loader (GRUB 2) security
- Disable unused software and services
- Understand and drop unnecessary capabilities for specific systemd units and the entire system
- Understand and configure Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP) and Exec-Shield
- Black and white list USB devices attached to a computer using USBGuard
- Create an SSH CA, create SSH certificates for host and user keys using the CA and configure OpenSSH to use SSH certificates
- Work with chroot environments
- Use systemd units to limit the system calls and capabilities available to a process
- Use systemd units to start processes with limited or no access to specific files and devices
- Use systemd units to start processes with dedicated temporary and /dev directories and without network access
- Understand the implications of Linux Meltdown and Spectre mitigations and enable/disable the mitigations
- Awareness of polkit
- Awareness of the security advantages of virtualization and containerization

## Configure BIOS and boot loader (GRUB 2) security:

To configure BIOS and bootloader security, you can take the following steps:

**Set a BIOS password:** Set a password to prevent unauthorized access to the BIOS settings.

**Enable secure boot:** Secure boot is a feature that checks that only trusted operating system and bootloader code is loaded at boot time. Enabling secure boot can help protect against malware and other attacks. Secure boot requires a UEFI BIOS.

Use a trusted bootloader: The GRUB 2 bootloader is commonly used in Linux systems. You can configure GRUB 2 to use Secure Boot and verify the signature of the kernel before booting.

Here's an example of how to configure GRUB 2 with Secure Boot on Ubuntu:

Install the necessary packages:

```
sudo apt-get install grub-efi-amd64-signed shim-signed
```

Edit the GRUB configuration file:

```
sudo nano /etc/default/grub
```

Add the following lines to the file:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX=""
GRUB_ENABLE_BLSCFG=true
```

The first two lines are optional and can be customized based on your system. The **GRUB_ENABLE_BLSCFG=true** line is necessary to enable the Boot Loader Specification (BLS) configuration file.

Update the GRUB configuration:

**sudo update-grub**

Install the GRUB bootloader to the EFI partition:

**sudo grub-install --target=x86_64-efi --efi-directory=/boot/efi --bootloader-id=ubuntu --recheck --no-floppy**

Reboot your system and verify that the bootloader is using Secure Boot by checking the secure boot status in the BIOS settings.

Disable unused software and services with example:

Disabling unused software and services can help reduce the attack surface of a system and improve its security. Here's an example of how to disable the Apache web server on Ubuntu:

Stop the Apache service:

**sudo systemctl stop apache2**

Disable the Apache service from starting at boot:

**sudo systemctl disable apache2**

Verify that the Apache service is disabled:

**sudo systemctl status apache2**

The output should show that the service is "disabled" and "inactive".

## Understand and drop unnecessary capabilities for specific systemd units and the entire system:

Capabilities are a feature in Linux that allow a process to perform specific privileged operations without needing full root access. However, some capabilities may not be necessary for certain processes or services, and can be a potential security risk. You can drop unnecessary capabilities for specific systemd units and the entire system using the following steps:

Identify the capabilities required by the systemd unit or service:

```
sudo systemctl show -p Capabilities myservicename.service
```

Identify any unnecessary capabilities that can be dropped.
Edit the systemd unit file to drop the unnecessary capabilities:

```
sudo nano /etc/systemd/system/myservicename.service
```

Add the following line to drop the unnecessary capabilities:

```
AmbientCapabilities=CAP_NET_BIND_SERVICE
```

Replace **CAP_NET_BIND_SERVICE** with the capability that you want to drop.

Reload the systemd daemon:

```
sudo systemctl daemon-reload
```

Restart the systemd unit or service:

```
sudo systemctl restart myservicename.service
```

**Other tools:**

**getcap and setcap:**

getcap and setcap are commands used to manage POSIX capabilities on Linux systems. POSIX capabilities are a way to grant specific privileges to individual processes, rather than granting full root-level access.

et's say you have a custom binary file called myprogram that needs to bind to a privileged port (port 80). Instead of running myprogram as the root user, you can use capabilities to grant it the CAP_NET_BIND_SERVICE capability, which allows it to bind to any port below 1024:

First, run **getcap /path/to/myprogram** to see if it already has any capabilities set. It will probably return an empty response.

Run **sudo setcap 'cap_net_bind_service=+ep' /path/to/myprogram** to grant it the necessary capability. Note that you need to run this command with sudo to have the required privileges to set capabilities.

Now, when you run **getcap /path/to/myprogram**, it should output **path/to/myprogram = cap_net_bind_service+ep**.

Finally, run myprogram, and it should be able to bind to port 80 without running as root.

**capsh:**

capsh is a command-line tool used to manipulate capabilities and their attributes on a running system. It can be used to inspect and modify capabilities of individual processes or the entire system. For example, running **capsh --print** will display the capabilities of the current shell session.

**sysctl and /etc/sysctl.conf:**

sysctl is a command-line tool used to view and modify kernel parameters at runtime. These parameters, also called "sysctl variables," can affect the behavior and performance of the Linux kernel.

**sysctl -w** to enable IPv4 forwarding:

Run sysctl net.ipv4.ip_forward to check the current value of net.ipv4.ip_forward. If the value is 0, IPv4 forwarding is currently disabled.

Use **sysctl -w net.ipv4.ip_forward=1** to set the net.ipv4.ip_forward parameter to 1, enabling IPv4 forwarding.

Run sysctl net.ipv4.ip_forward again to verify that the new value has been set.

After following these steps, IPv4 forwarding should be enabled on your system. Note that this change is temporary and will be lost after a system reboot unless you modify the /etc/sysctl.conf file to make it permanent.

/etc/sysctl.conf is a configuration file used to set sysctl variables at boot time. The file contains key-value pairs, where the key is the name of the variable and the value is the desired setting. For example, adding the line **net.ipv4.tcp_syncookies=1** to **/etc/sysctl.conf** will enable TCP syncookies to protect against SYN flooding attacks.

To apply changes made in /etc/sysctl.conf, you can either reboot the system or run the sysctl command with the -p option to reload the file. For example, running **sysctl -p** will apply any changes made to /etc/sysctl.conf.

**Understand and configure Address Space Layout Randomization (ASLR):**

ASLR is a security feature in Linux that helps prevent buffer overflow attacks by randomizing the memory layout of processes. This makes it more difficult for attackers to predict

**Data Execution Prevention (DEP)**

DEP is a security feature in Windows operating systems that helps prevent malicious code from executing in memory by marking certain areas of memory as non-executable. This means that code cannot be run from those areas, making it harder for attackers to exploit vulnerabilities in the system. DEP is enabled by default on most modern versions of Windows and can be configured in the system settings.

**NX bit**

The NX bit (No-eXecute bit) is a feature in computer processors that allows marking of certain areas of memory as non-executable. This helps prevent malicious code from being executed by marking the memory as read-only, making it harder for attackers to exploit vulnerabilities in the system. The NX bit was introduced by Intel in 2004 and is now commonly used by most modern processors. It is often used in combination with other security features, such as Data Execution Prevention (DEP), to provide additional protection against malware and other malicious attacks.

**Exec-Shield**

Exec-Shield is a security feature found in some Linux kernels that helps prevent buffer overflow attacks. It works by creating a layout of virtual memory that makes it harder for attackers to exploit vulnerabilities in the system. This is achieved by marking certain areas of memory as non-executable and randomizing the location of stack and heap memory, making it more difficult for an attacker to control the program execution. Exec-Shield is an optional feature that can be enabled or disabled depending on the security requirements of the system. It is typically used in servers, critical systems, and other high-security environments to enhance protection against malicious attacks.

In summary, DEP and NX bit provide similar protection against malicious code execution by marking certain areas of memory as non-executable, while Exec-Shield provides additional protection for Linux systems against buffer overflow attacks.

**USBGuard**

USBGuard is a Linux-based software that can be used to control access to USB devices attached to a computer. It allows you to create a blacklist or whitelist of devices, so that you can either block access to specific devices or only allow access to a select set of devices.

A blacklist is a list of devices that are blocked and cannot be accessed by the system, while a whitelist is a list of devices that are allowed and can be accessed. When using USBGuard, you can configure the policy so that only devices on the whitelist are allowed to be attached, and all other devices are automatically blocked.

By using USBGuard, you can increase the security of your computer by preventing unauthorized access to sensitive data or potentially malicious devices. You can also use it to ensure that only approved devices are used for specific tasks, such as scanning documents or printing.

Note: USBGuard is a third-party software, and its usage and configuration may vary depending on the system it is being used on.

Here's an example of using USBGuard to create a whitelist of devices:

Install USBGuard on your Linux system.

Identify the unique identifier (ID) of the USB devices you want to allow access to. This can be done using the **lsusb** command in the terminal.

Create a new rule in the USBGuard configuration file, which is typically located at **/etc/usbguard/usbguard-daemon.conf**, to allow access to the devices with the specified IDs.

Example:

```
# Allow access to a specific USB device
allow with-interface {
  bInterfaceClass == 8
  bInterfaceSubClass == 6
  bInterfaceProtocol == 80
  bDeviceClass == 255
  bDeviceSubClass == 255
  bDeviceProtocol == 255
}
```

Restart the USBGuard service to apply the changes.

With this configuration, only devices that match the specified interface and device class, sub-class, and protocol will be allowed access to the system, while all other devices will be automatically blocked.

**What about the /etc/usbguard/rules.conf file?**

The /etc/usbguard/rules.conf file is a configuration file used by USBGuard to store rules for controlling access to USB devices. The rules in this file determine which devices are allowed or blocked when they are attached to the system. The rules in this file are evaluated in order, and the first rule that matches a device is applied.

**Create an SSH CA, create SSH certificates for host and user keys using the CA and configure OpenSSH to use SSH certificates:**

To create an SSH certificate authority (CA) and generate host and user certificates, you can use the **ssh-keygen** utility that comes with OpenSSH. Here are the steps to accomplish this:

## Server side configuration:

**Create the CA key:**

```
$ ssh-keygen -f ssh_ca
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ssh_ca.
Your public key has been saved in ssh_ca.pub.
The key fingerprint is:
```

```
SHA256:UjJ9Lg3W8YvJFijbSZB0nI/Qn8LkRzR4mDX4l9EMB1g user@host
The key's randomart image is:
+---[RSA 2048]----+
|                 |
| .               |
|   o             |
|    = .          |
|   + S +         |
| .   = +         |
|. . .   .        |
|. . . . .        |
+----[SHA256]-----+
```

**Create the host key:**

```
$ ssh-keygen -f ssh_host
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ssh_host.
Your public key has been saved in ssh_host.pub.
The key fingerprint is:
SHA256:UjJ9Lg3W8YvJFijbSZB0nI/Qn8LkRzR4mDX4l9EMB1g user@host
The key's randomart image is:
+---[RSA 2048]----+
|                 |
| .               |
|   o             |
|    = .          |
|   + S +         |
| .   = +         |
|. . .   .        |
|.....      |
+----[SHA256]-----+
```

**Client side configuration:**

**Create the user key:**

```
$ ssh-keygen -f ssh_user
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ssh_user.
Your public key has been saved in ssh_user.pub.
The key fingerprint is:
SHA256:UjJ9Lg3W8YvJFijbSZB0nI/Qn8LkRzR4mDX4l9EMB1g user@host
The key's randomart image is:
+---[RSA 2048]----+
|                 |
| .               |
|  o              |
|   = .           |
|  + S +          |
| .  = +          |
|...  .           |
+----[SHA256]-----+
```

**Sign the host and user keys with the CA:**

```
$ ssh-keygen -s ssh_ca -I host_key -h -n host ssh_host.pub
Signature done: RSA (SHA256)
Cert
```

**Configure openssl to use it on server Side:**

To configure OpenSSH to use SSH certificates, you'll need to modify the OpenSSH server and client configuration files.

Modify the OpenSSH server configuration file, usually located at /etc/ssh/sshd_config, and add the following lines:

```
HostKey /path/to/ssh_host
TrustedUserCAKeys /path/to/ssh_ca.pub
AuthorizedKeysFile /path/to/authorized_keys_%u
```

Modify the OpenSSH client configuration file on the client side, usually located at /etc/ssh/ssh_config, and add the following line:

```
IdentityFile /path/to/ssh_user
```

Restart the OpenSSH server to apply the changes:

```
$ sudo service ssh restart
```

Copy the CA public key to the client machine:

```
$ scp user@server:/path/to/ssh_ca.pub .
```

Add the CA public key to the client machine's trusted CA list:

```
$ cat ssh_ca.pub >> ~/.ssh/authorized_keys
```

Connect to the server using the client certificate:

```
$ ssh -i /path/to/ssh_user user@server
```

Note: The steps outlined above are just a basic example and may need to be modified for your specific use case and security requirements.

**Work with chroot environments:**

Working with chroot environments allows you to run a process with a different root directory than the one it would normally use. This is useful for a number of reasons, such as isolating an application from the rest of the system or providing a clean environment for software testing.

Here's a basic example of how to set up and use a chroot environment:

1. Create a directory that will serve as the root directory for the chroot environment. For example, **mkdir /chrootenv**.
2. Copy the necessary files and libraries into the chroot directory. This typically involves copying the binaries for the applications you want to run, as well as any shared libraries they depend on. For example, if you want to run the **ls command** in the chroot environment, you would need to copy the ls binary and any libraries it depends on, such as **libc.so.6**.
3. Use the chroot command to run a command within the chroot environment. For example, **chroot /chrootenv ls** would run the ls command within the chroot environment.

Here's an example of how to use a chroot environment to run an Apache web server:

1. Create a directory for the chroot environment, such as **mkdir /apachechroot**.
2. Copy the necessary files into the chroot directory. For Apache, this typically involves copying the **httpd binary**, any configuration files, and any necessary shared libraries.
3. Copy any required files or directories into the chroot environment. For example, if your Apache configuration files reference files in /var/www, you would need to copy those files into the chroot environment as well.
4. Set up the necessary networking configuration. You may need to copy **/etc/resolv.conf** and **/etc/hosts** into the chroot environment to ensure that DNS and hostname resolution work correctly.
5. Use the chroot command to run Apache within the chroot environment. For example, **chroot /apachechroot /usr/sbin/httpd -f /path/to/httpd.conf** would start Apache using the configuration file at /path/to/httpd.conf within the chroot environment.

Note that working with chroot environments can be complex and requires a good understanding of how file systems work. It's important to carefully choose what files and libraries you include in the chroot environment to ensure that everything your application needs is available, but also to avoid including unnecessary files that could potentially cause problems.

## Use systemd units to limit the system calls and capabilities available to a process:

Systemd provides a mechanism to limit the system calls and capabilities available to a process using systemd units. A systemd unit is a configuration file that defines the properties of a service, including the environment in which the service runs and any resource constraints.

Create a new systemd unit file for the process. This file should have a .service extension and can be placed in the /etc/systemd/system/ directory.

Add the following lines to the unit file to limit the system calls available to the process:

```
[Service]
SystemCallFilter=@system-service
```

Add the following lines to the unit file to limit the capabilities available to the process:

```
[Service]
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
```

Replace CAP_NET_BIND_SERVICE with the desired capabilities. The list of available capabilities can be found in the /usr/include/linux/capability.h file.

Save and close the unit file.

Reload the systemd configuration using the following command:

```
sudo systemctl daemon-reload
```

Start the service defined in the unit file using the following command:

```
sudo systemctl start [service name]
```

With this configuration, the process will run with the specified system call filter and capability bounding set, limiting the system calls and capabilities available to it. This can help improve the security of the process and the system as a whole.

## Use systemd units to start processes with limited or no access to specific files and devices

Systemd provides a mechanism to start processes with limited or no access to specific files and devices using systemd units. A systemd unit is a configuration file that defines the properties of a service, including the environment in which the service runs and any resource constraints.

**Here's how to start processes with limited or no access to specific files and devices using systemd units:**

Create a new systemd unit file for the process. This file should have a .service extension and can be placed in the /etc/systemd/system/ directory.

Add the following lines to the unit file to restrict access to specific files and devices:

```
[Service]
ReadWritePaths=-/path/to/restricted/file
ReadWritePaths=-/dev/restricted_device
```

Replace /path/to/restricted/file and /dev/restricted_device with the specific files and devices that you want to restrict access to.

Save and close the unit file.

Reload the systemd configuration using the following command:

```
sudo systemctl daemon-reload
```

Start the service defined in the unit file using the following command:

```
sudo systemctl start [service name]
```

With this configuration, the process will start with limited or no access to the specified files and devices, improving the security of the process and the system as a whole.

## Linux Meltdown and Spectre Mitigations:

Meltdown and Spectre are vulnerabilities that allow attackers to exploit a computer's microprocessor to read sensitive information from memory. To mitigate these vulnerabilities, Linux introduced kernel patches that i
mplement various mitigation techniques.

These mitigations can have performance impacts, so it is important to understand the implications of enabling or disabling them. For example, disabling the mitigations can improve performance but also increase the risk of exploitation.

To enable or disable the mitigations, you can use kernel command-line parameters or system configuration options. These can vary depending on the Linux distribution and version you are using. In general, it is recommended to consult the documentation and support resources for your specific distribution and version.

## Awareness of Polkit:

Polkit is a system service that provides a way for unprivileged processes to request privileged operations. It is commonly used on Linux systems to manage access to system resources and functionality.

As an administrator or user, it is important to be aware of Polkit and how it is used in your system. By default, Polkit is configured to grant access to certain operations to certain users or groups. You can modify these settings to restrict or expand access as needed.

To manage Polkit settings, you can use configuration files, command-line tools, or graphical interfaces depending on your system and distribution. For example, in Ubuntu, you can use the "Users and Groups" tool in the System Settings to manage Polkit access.

**Awareness of Virtualization and Containerization Security Advantages:**

Virtualization and containerization are technologies that enable the creation of isolated environments for running applications and services. These technologies provide several security advantages, such as:

Isolation: Virtual machines and containers can be isolated from the host and other virtual machines or containers, which can help prevent the spread of malware or attacks.

Sandboxing: Virtual machines and containers can be configured with restricted access to system resources and functionality, which can limit the impact of a successful attack.

Live Migration: Virtual machines and containers can be migrated between hosts without downtime, which can enable better disaster recovery and high availability.

However, it is important to be aware that virtualization and containerization can also introduce new security challenges. For example, vulnerabilities in the virtualization or containerization software itself can potentially compromise the security of the entire system. Additionally, managing and securing a large number of virtual machines or containers can be complex and require specialized knowledge and tools.

## 332.2 Host Intrusion Detection (weight: 5)

- Use and configure the Linux Audit system
- Use chkrootkit

- Use and configure rkhunter, including updates
- Use Linux Malware Detect
- Automate host scans using cron
- Use RPM and DPKG package management tools to verify the integrity of installed files
- Configure and use AIDE, including rule management
- Awareness of OpenSCAP

**Use and configure the Linux Audit system:**

The Linux Audit system allows system administrators to track security-relevant events on their system. Some of the events that can be tracked include system calls, file access, and network activity. The audit system is configured through the auditd daemon, which writes audit records to a designated log file. The **auditctl command** is used to manage rules for the audit system, and the ausearch and aureport commands are used to search and generate reports on audit log files.

Sure, here are the explanations of each tool with examples:

**auditctl** used to specify what events should be audited and written to the audit log. Here's an example command to add a rule to monitor all file deletions:

```
sudo auditctl -w /var/log/audit/ -p wa -k delete
```

**ausearch, aureport:**

These are two command-line tools that are used to search and report on audit log files. ausearch allows you to search the audit log files based on various criteria, such as date range, event type, and user. aureport is used to generate reports based on the audit log data. Here's an example command to search for all events related to the "sudo" command:

```
sudo ausearch -c sudo
```

**auditd.conf:**

auditd.conf is the main configuration file for the auditd daemon. It contains settings such as log location, max log file size, and the audit rules directory. Here's an example of modifying the log file location:

```
sudo vi /etc/audit/auditd.conf
```

```
# Modify the following line
```

```
log_file = /var/log/audit/audit.log
```

**audit.rules:**

audit.rules is the file where you can define custom audit rules to monitor various events. Here's an example of a rule to monitor all changes to the "/etc/shadow" file:

```
-w /etc/shadow -p wa -k shadow_changes
```

**pam_tty_audit.so:**

pam_tty_audit.so is a PAM module that allows administrators to log all input and output to a TTY. This can be useful for monitoring and auditing user activity. Here's an example of how to enable this module:

```
sudo vi /etc/pam.d/login
```

```
# Add the following line to the end of the file
```

```
session required pam_tty_audit.so enable=*
```

**Use chkrootkit:**

chkrootkit is a tool that checks for signs of rootkits on a Linux system. Rootkits are malicious software that allow attackers to gain unauthorized access to a system and remain undetected. chkrootkit scans the system for common rootkit signatures and can also check for suspicious kernel modules and hidden files.

```
sudo chkrootkit
```

**Use and configure rkhunter, including updates:**

rkhunter is another tool for detecting rootkits on a Linux system. It checks for suspicious files and directories, suspicious kernel modules, and other indicators of a compromised system. rkhunter is configured through the /etc/rkhunter.conf file, which specifies which tests to run and what directories to scan. It is important to keep rkhunter up-to-date to ensure that it is checking for the latest rootkit signatures.

```
sudo rkhunter --update

sudo rkhunter --check
```

**/etc/rkhunter.conf:**

/etc/rkhunter.conf is the main configuration file for rkhunter. It contains settings such as the directories to scan and the frequency of scans. Here's an example of modifying the directories to scan:

```
sudo vi /etc/rkhunter.conf

# Modify the following line

ALLOWED_DIRS=/var/log /var/run /var/spool
```

**Use Linux Malware Detect:**

Linux Malware Detect (LMD) is a malware scanner for Linux systems. It scans files for known malware signatures and can also detect hidden files and obfuscated code. LMD is configured through the conf.maldet file, which specifies which directories to scan and what actions to take when malware is detected.

Example:

To scan a directory with maldet, use the following command:

**maldet -a /path/to/directory**

conf.maldet:

The configuration file for maldet is located at /usr/local/maldetect/conf.maldet. This file can be used to configure various options for the malware scanner, such as email notifications and quarantine settings.

Example:

To enable email notifications, open /usr/local/maldetect/conf.maldet and set the following options:

**email_alert=1**

**email_subj="Malware alert for $(hostname) - $(date +%Y-%m-%d)"**

**email_addr="youremail@example.com"**

**Automate host scans using cron:**

cron is a Linux utility that allows tasks to be scheduled to run at specified times. By using cron, system administrators can automate regular malware scans and other security tasks. For example, a cron job could be created to run rkhunter or LMD scans every night at 2 am.

**Use RPM and DPKG package management tools to verify the integrity of installed files:**

RPM and DPKG are package management tools used on Red Hat-based and Debian-based Linux systems, respectively. Both tools allow system administrators to verify the integrity of installed packages and their associated files. This is important for ensuring that system files have not been tampered with and that the system is secure.

**check checksum of package to verfiy it with rpm and dpkg**

To check the checksum of a package with rpm, use the -K or --checksig option followed by the path to the RPM file. For example:

**rpm -K package.rpm**

This will check the digital signature of the package and display the results.

To check the checksum of a package with dpkg, use the --verify option followed by the name of the package. For example:

**dpkg --verify package-name.deb**

This will verify the integrity of the package and display any discrepancies in the checksums of the files in the package.

**Configure and use AIDE, including rule management:**

AIDE (Advanced Intrusion Detection Environment) is a tool for detecting file system changes on a Linux system. It creates a database of file properties, including permissions, ownership, and checksums, and then compares that database to the current state of the file system to detect changes. AIDE is configured through the **/etc/aide/aide.conf** file, which specifies which directories and files to monitor and what actions to take when changes are detected.

**Here is an example of the /etc/aide/aide.conf configuration file:**

```
# AIDE configuration file

@@define DBDIR /var/lib/aide
```

```
database=file:/var/lib/aide/aide.db
database_out=file:/var/lib/aide/aide.db.new
gzip_dbout=yes

# List of directories and files to be checked
/etc
/bin
/sbin
/usr/bin
/usr/sbin
/usr/local/bin
/usr/local/sbin
/var/log

# List of regular expressions for files to be excluded from checking
!/@tmp
!/var/log.*
!/var/spool/at
!/var/spool/cron
!/var/spool/exim
!/var/spool/mail
!/var/spool/squid
!/var/spool/samba
!/var/spool/postfix
!/var/spool/abrt
!/var/spool/sudo
!/var/run/screen
!/var/cache/dnf
!/var/cache/yum
```

```
# Rule configuration
ALLXTRAHASHES = sha512
ALLPERMS = p+i+u+g+acl+selinux
```

**Awareness of OpenSCAP:**

OpenSCAP is a security compliance tool that provides a framework for managing security policies and auditing compliance on Linux systems. It includes a variety of tools for scanning, monitoring, and reporting on security configurations, and supports a wide range of security benchmarks and guidelines. OpenSCAP can be used to assess compliance with standards like HIPAA, PCI DSS, and NIST SP 800-53.

## 332.3 Resource Control (weight: 3)

- Understand and configure ulimits
- Understand cgroups, including classes, limits and accounting
- Manage cgroups and process cgroup association
- Understand systemd slices, scopes and services
- Use systemd units to limit the system resources processes can consume
- Awareness of cgmanager and libcgroup utilities

### Ulimit

**The ulimits** configuration is a mechanism used to limit the resources that a user or process can consume on a Unix-based system. This is done by setting various limits, such as the maximum number of open file descriptors, the maximum size of a process's data segment, or the maximum number of processes that a user can run simultaneously.

The ulimits configuration can be set in two ways: through the /etc/security/limits.conf file, or through the ulimit command.

**/etc/security/limits.conf file**: The **/etc/security/limits.conf** file is a global configuration file that sets limits for all users on the system. The file uses the following syntax:

**<user> <soft limit> <hard limit>**

For example, to set the maximum number of open file descriptors for the user "testuser" to 1024, the following line can be added to the /etc/security/limits.conf file:

**testuser soft nofile 1024**

ulimit command: The ulimit command can be used to set limits for the current shell session or for a specific process. The syntax for the ulimit command is as follows:

- **ulimit -<option> <limit>**
- **ulimit -a**: This option displays all of the current resource limits for the current shell session.
- **ulimit -c:** This option displays or sets the limit on the size of core files that can be generated.
- **ulimit -f:** This option displays or sets the limit on the maximum size of files that can be created.
- **ulimit -n:** This option displays or sets the limit on the number of open file descriptors.
- **ulimit -u**: This option displays or sets the limit on the number of processes that can be created for a user.
- **ulimit -v:** This option displays or sets the limit on the amount of virtual memory that can be consumed by a process.

For example, to set the maximum number of open file descriptors for the current shell session to 1024, the following command can be used:

```
ulimit -n 1024
```

It is important to note that ulimits should be set with care, as setting limits too low can prevent a user or process from functioning properly, while setting limits too high can result in excessive resource usage and potential security issues.

## pam_limit

**pam_limits.so** is a PAM (Pluggable Authentication Modules) module that allows administrators to set limits on system resources, such as the maximum number of open file descriptors or the maximum size of core files, for individual users or groups of users.

pam_limits.so is typically used in conjunction with the /etc/security/limits.conf file, which specifies the resource limits for different users or groups. When a user logs in, pam_limits.so reads the limits specified in the /etc/security/limits.conf file and sets the appropriate resource limits for the user's session.

For example, if the following line is present in the **/etc/security/limits.conf** file:

**@users soft nofile 1024**

This line specifies that all users in the "users" group should have a soft limit of 1024 on the maximum number of open file descriptors. When a user in the "users" group logs in, pam_limits.so will read this line and set the user's soft limit on open file descriptors to 1024.

In order to use pam_limits.so, it must be included in the appropriate PAM configuration file, such as **/etc/pam.d/system-auth or /etc/pam.d/login**. The exact configuration will depend on the specific Linux distribution being used.

## Cgroups

**Cgroups, or control groups**, is a Linux kernel feature that allows administrators to manage and allocate system resources, such as CPU, memory, and I/O bandwidth, for groups of processes. This can be useful for enforcing resource quotas and prioritizing system resources for different workloads.

Cgroups consist of the following components:

- **Classes:** Classes define the different types of resources that can be managed by cgroups, such as CPU, memory, and I/O bandwidth.
- **Limits:** Limits specify the maximum amount of each resource that can be consumed by a cgroup. For example, administrators can specify a limit on the maximum amount of memory that a cgroup can consume.

- **Accounting:** Accounting refers to the measurement and reporting of resource usage by cgroups. This information can be used to track resource consumption and identify any performance bottlenecks.

Cgroups can be managed using the cgroup utilities, such as cgcreate, cgset, and cgexec. For example, the following commands can be used to create a cgroup for a specific application and set a limit on the maximum amount of memory that it can consume:

```
# Create a cgroup for the application
sudo cgcreate -g memory:/app

# Set a limit of 512 MB on the maximum amount of memory that the cgroup can consume
sudo cgset -r memory.limit_in_bytes=512M /app

# Launch the application in the cgroup
sudo cgexec -g memory:/app ./application
```

By using cgroups, administrators can enforce resource quotas and prioritize system resources for different workloads, leading to improved system performance and stability.

Managing cgroups involves creating, modifying, and deleting control groups, as well as associating processes with cgroups. The following are the steps to manage cgroups and process cgroup association:

**Create a cgroup:** The cgcreate command can be used to create a new cgroup. For example, to create a cgroup named "app" under the "cpu" class, you can use the following command:

**sudo cgcreate -g cpu:/app**

**Modify cgroup limits:** The cgset command can be used to set limits on the resources that a cgroup can consume. For example, to limit the CPU usage of a cgroup named "app" to 50% of a single core, you can use the following command:

```
sudo cgset -r cpu.cfs_quota_us=50000 /app
```

**Associate a process with a cgroup: T**he cgexec command can be used to launch a process in a specific cgroup. For example, to launch the firefox application in the "app" cgroup, you can use the following command:

```
sudo cgexec -g cpu:/app firefox
```

Alternatively, you can also use the echo command to associate an existing process with a cgroup. For example, to associate process ID 1234 with the "app" cgroup, you can use the following command:

```
sudo echo 1234 > /sys/fs/cgroup/cpu/app/tasks
```

**Monitor cgroup usage:** The **cgtop** command can be used to monitor the resource usage of cgroups in real-time.

**Delete a cgroup:** The cgdelete command can be used to delete a cgroup. For example, to delete the "app" cgroup, you can use the following command:

```
sudo cgdelete cpu:/app
```

By managing cgroups and associating processes with cgroups, you can effectively control and allocate system resources, such as CPU, memory, and I/O bandwidth, to improve system performance and stability.

## /proc/cgroups

/proc/cgroups is a virtual file system located in the Linux proc file system that provides information about the control groups (cgroups) supported by the system. The /proc/cgroups file contains information about the various subsystems (memory, CPU, etc.) and the hierarchical structure of cgroups in the system.

Each line in the **/proc/cgroups** file represents a subsystem and contains the following information:

- **Subsystem name:** The name of the cgroup subsystem.
- **Hierarchy ID:** The ID of the hierarchy to which the subsystem belongs.
- **Number of cgroups:** The number of cgroups present in the hierarchy.
- **Number of enabled cgroups:** The number of enabled cgroups in the hierarchy.
- **Copy-on-write:** Indicates if the subsystem uses copy-on-write (CoW) for child cgroups.
- **Control file enabled:** Indicates if the subsystem provides control files for the cgroups.

For example, the following line in the /proc/cgroups file shows information about the memory subsystem:

```
#subsys_name   hierarchy   num_cgroups   enabled
memory  2      1           1
```

The information in the /proc/cgroups file can be used to determine the cgroup subsystems supported by the system and the hierarchical structure of cgroups. This information can be useful in understanding the behavior of cgroups and how they allocate system resources.

To list the available control groups (cgroups) in a Linux system, you can use the following command:

```
# cat /proc/cgroups
```

This command will show the names of the cgroup subsystems, the hierarchy IDs, the number of cgroups, and other information about the cgroup hierarchies.

Alternatively, you can use the following command to view a list of the available cgroups in a tree-like format:

```
lscgroup
```

This command will show the cgroups and their hierarchical structure, including the parent and child cgroups, their subsystems, and their associated processes.

## Systemd Slice, Scops and Service

**Systemd** is a system and service manager for Linux. It provides a unified way to manage the startup, shutdown, and overall management of various services and applications in a Linux system.

In systemd, there are three important concepts:

- **systemd slices:** systemd slices provide a way to divide a system into multiple logical parts, each with its own set of resources and priorities. Slices allow administrators to manage system resources and prioritize the allocation of these resources to different parts of the system.
- **systemd scopes:** systemd scopes are similar to slices, but they are specifically used for managing groups of processes that are dynamically created and managed by systemd. Scopes allow administrators to manage and prioritize dynamically created processes in a unified way.
- **systemd services:** systemd services are units of work that run in the background and provide specific functions to the system. Services are responsible for starting, stopping, and managing applications and other background processes.

Each systemd service is defined in a unit file, which specifies the service's name, description, dependencies, and other configuration options. The unit files are located in the **/etc/systemd/system** directory and can be managed using the systemctl command.

By understanding the concepts of systemd slices, scopes, and services, administrators can effectively manage the resources and processes of a Linux system using systemd.

### Here's an example of using systemd slices, scopes, and services:

Suppose you have a Linux server that runs multiple services, including a web server, a database server, and a mail server. To manage the resources and priorities of these services, you can create a systemd slice for each of them. For example, you can create a slice named web.slice, a slice named db.slice, and a slice named mail.slice.

Within each slice, you can define the resources and priorities for each service. For example, you can specify that the **web.slice** should have a higher priority than the **db.slice** and the **mail.slice**. You can also specify the maximum amount of memory, CPU, and other resources that each slice can use.

To manage the services within each slice, you create a systemd service for each one. For example, you can create a service named **web.service**, a service named **db.service**, and a service named **mail.service**.

The services can then be started, stopped, and managed using the systemctl command. For example, to start the web server, you can run the following command:

```
# systemctl start web.service
```

To check the status of the services, you can use the systemctl command with the status option. For example:

```
# systemctl status web.service
```

To manage the resources of each slice, you can use the **systemd-cgtop** command to view the resource usage of each slice in real-time.

**systemd-cgls** displays the control groups and their associated processes in a hierarchical tree-like structure, with the root control group at the top and child control groups below. The output of the command shows the control group name, hierarchy ID, number of processes, and other information about the control group.

By using systemd slices, scopes, and services, you can effectively manage the resources and processes of your Linux server and ensure that each service has the resources and priority it needs to function properly.

## Systemd units

**systemd units** can be used to limit the system resources that processes can consume by specifying resource limits in the unit file of a systemd service. The following are the most commonly used resource limits:

- **MemoryLimit:** Specifies the maximum amount of memory that a service can consume. The limit is specified in bytes and can be set with the MemoryLimit= directive in the unit file.
- **CPUShares:** Specifies the relative CPU time that a service should receive compared to other services. The higher the number, the more CPU time the service will receive. The limit is specified as a positive integer and can be set with the **CPUShares=** directive in the unit file.
- **BlockIOWeight:** Specifies the relative weight of a service for disk I/O operations. The higher the weight, the more disk I/O operations the service will receive. The limit is specified as a positive integer and can be set with the **BlockIOWeight=** directive in the unit file.

- **Nice:** Specifies the priority level of a service for CPU scheduling. The higher the nice value, the lower the priority of the service. The limit is specified as an integer between -20 and 19 and can be set with the **Nice=** directive in the unit file.

By using these resource limits, you can ensure that a service does not consume more resources than it needs and that the system remains stable and responsive.
Here's an example of how to limit the memory usage of a service in a systemd unit file:

```
[Service]
MemoryLimit=512M
```

In this example, the **MemoryLimit** is set to 512 MB, which means that the service cannot consume more than 512 MB of memory.
You can apply these limits by creating or modifying a unit file for the service in the **/etc/systemd/system** directory, and then reloading the systemd configuration with the following command:

```
# systemctl daemon-reload
```

## cgmanager and libcgroup

**cgmanager and libcgroup** are Linux utilities for managing and interacting with control groups (cgroups). **cgmanager** is a daemon that provides a high-level interface to the control group functionality in the Linux kernel. It allows you to create, manage, and remove control groups, as well as assign processes to them. **cgmanager** uses the **libcgroup** library to communicate with the Linux kernel and manage control groups. **libcgroup** is a library that provides a low-level interface to the control group functionality in the Linux kernel. It provides a simple and efficient way for applications to interact with control groups, including creating, modifying, and removing control groups and assigning processes to them.
Using these utilities, you can control the allocation and management of system resources, such as CPU, memory, and I/O bandwidth, among groups of processes. This can be useful in scenarios where you need to manage resource allocation and limit the resources that a particular process or group of processes can consume.

Here's an example of how to use cgmanager to create a new control group and assign a process to it:

```
# cgcreate -g cpu:/mygroup
# cgexec -g cpu:mygroup <command>
```

In this example, the first command creates a new control group named mygroup under the cpu hierarchy. The second command runs the specified <command> in the context of the mygroup control group, so that the process will be assigned to the control group and will be subject to the resource limits specified for that group.

# *Topic 333: Access Control*

## 333.1 Discretionary Access Control (weight: 3)

- Understand and manage file ownership and permissions, including SetUID and SetGID bits
- Understand and manage access control lists
- Understand and manage extended attributes and attribute classes

## File Ownership and Permissions:

- **File ownership:** Refers to the user and group that own a file or directory.
- **Permissions:** Determine who can access and modify a file or directory. Permissions are broken down into three categories: owner, group, and other.
- **chmod:** Command used to change permissions on files and directories.
- **chown:** Command used to change the ownership of a file or directory.
- **chgrp:** Command used to change the group ownership of a file or directory.
- **SetUID:** A special permission that allows a user to execute a file with the permissions of the file's owner.
- **SetGID:** A special permission that allows a user to execute a file with the permissions of the file's group owner.

## Access Control Lists:

- **ACLs:** Provide a way to give more fine-grained control over file and directory permissions than traditional Unix permissions.
- **getfacl:** Command used to display the ACLs for a file or directory. Example: **getfacl file.txt**
- **setfacl:** Command used to set the ACLs for a file or directory. Example: **setfacl -m u:user:rwx file.txt**

## Extended Attributes and Attribute Classes:

- **Extended attributes:** Additional metadata that can be attached to files and directories.
- **Attribute classes:** Categories that extended attributes can be grouped into.
- **getfattr:** Command used to display the extended attributes for a file or directory. Example: **getfattr -d file.txt**

- setfattr: Command used to set the extended attributes for a file or directory. Example: **setfattr -n user.myattr -v "my value" file.txt**

## 333.2 Mandatory Access Control (weight: 5)

- Understand the concepts of type enforcement, role based access control, mandatory access control and discretionary access control
- Configure, manage and use SELinux
- Awareness of AppArmor and Smack

**Type enforcement:** A security mechanism that ensures that only specific types of processes can access certain types of resources.

**Role-based access control (RBAC):** A security model in which access is based on a user's assigned role, rather than their identity.

**Mandatory access control (MAC):** A security model in which access to resources is based on rules that are defined by a central authority, rather than by individual users or applications.

**Discretionary access control (DAC):** A security model in which access to resources is based on the permissions granted by the resource owner.

### SELinux:

**SELinux:** A Linux security module that implements MAC.

**getenforce:** Command used to display the current SELinux mode. Example: **getenforce**

**setenforce:** Command used to change the current SELinux mode. Example: **setenforce 1**

**selinuxenabled:** Command used to check if SELinux is enabled. Example: **selinuxenabled**

**getsebool:** Command used to display the current value of a SELinux boolean. Example: **getsebool httpd_can_network_connect**

**setsebool:** Command used to set the value of a SELinux boolean. Example: **setsebool httpd_can_network_connect 1**

**togglesebool:** Command used to toggle the value of a SELinux boolean. Example: **togglesebool httpd_can_network_connect**

**fixfiles:** Command used to restore the SELinux context of a file or directory. Example: **fixfiles restore /var/www**

**restorecon:** Command used to restore the SELinux context of a file or directory. Example: **restorecon /var/www/html**

**setfiles:** Command used to set the SELinux context of a file or directory. Example: **setfiles /usr/share/selinux/default.pp**

**newrole:** Command used to switch to a new SELinux role. Example: **newrole staff_r**

**setcon**: Command used to set the SELinux context of a process. Example: **setcon unconfined_u:system_r:httpd_t:s0**

**runcon:** Command used to run a command with a specified SELinux context. Example: **runcon unconfined_u:system_r:httpd_t:s0 /usr/bin/php**

**chcon:** Command used to change the SELinux context of a file or directory. Example: **chcon system_u:object_r:httpd_sys_content_t:s0 /var/www/html**

**semanage:** Command used to manage SELinux policy modules, ports, users, and more. Example: **semanage port -a -t http_port_t -p tcp 8080**

**sestatus:** Command used to display the current SELinux status, mode, and policy. Example: **sestatus**

**seinfo:** Command used to display information about SELinux policy types and attributes. Example: **seinfo -t**

**apol:** Graphical tool used to create and edit SELinux policy modules.

**seaudit**: Command used to view SELinux audit logs. Example: **seaudit -a /var/log/audit/audit.log**

**audit2why**: Command used to convert SELinux audit logs into human-readable messages.
Example: **audit2why < /var/log/audit/audit.log**

**audit2allow**: Command used to create new SELinux policy modules based on audit logs.
Example: **audit2allow -a -M mypolicy < /var/log/audit/audit.log**

**/etc/selinux/*:** Directory containing SELinux configuration files

**AppArmor and Smack** are also Linux security modules that provide access control mechanisms for processes and files.

**AppArmor:**

- AppArmor is a Linux security module that provides mandatory access control (MAC) by restricting the actions that a process can perform based on the process's profile.
- AppArmor profiles can be created and customized for individual applications or services, and these profiles define what files, directories, network ports, and other system resources the process is allowed to access.
- AppArmor profiles can be managed and configured using the apparmor command-line tool.

**Smack:**

- Smack is a Linux security module that provides mandatory access control (MAC) based on labels that are associated with files and processes.
- Smack labels are used to define access control policies that restrict which processes can access which files, directories, and other system resources.
- Smack labels can be managed and configured using the **smackctl** command-line tool.

While SELinux, AppArmor, and Smack are all Linux security modules that provide mandatory access control, they differ in their approaches and mechanisms for implementing access control policies.

## *Topic 334: Network Security*

### 334.1 Network Hardening (weight: 4)

- Understand wireless networks security mechanisms
- Configure FreeRADIUS to authenticate network nodes
- Use Wireshark and tcpdump to analyze network traffic, including filters and statistics
- Use Kismet to analyze wireless networks and capture wireless network traffic
- Identify and deal with rogue router advertisements and DHCP messages
- Awareness of aircrack-ng and bettercap

**Understand wireless networks security mechanisms:**

- Knowledge of the security mechanisms used in wireless networks, such as WPA, WPA2, and WPA3.
- Knowledge of the vulnerabilities and attacks that can affect wireless networks, such as rogue access points, eavesdropping, and denial-of-service attacks.

**Configure FreeRADIUS to authenticate network nodes:**

- FreeRADIUS is an open source RADIUS server that can be used to authenticate network nodes, such as wireless clients and VPN users.
- Knowledge of how to configure FreeRADIUS to authenticate network nodes using different authentication methods, such as EAP-TLS and PEAP.

**Use Wireshark and tcpdump to analyze network traffic, including filters and statistics:**

- Wireshark and tcpdump are network packet analyzers that can capture and analyze network traffic.
- Knowledge of how to use these tools to capture and analyze network traffic, including how to apply filters and view statistics.

**Use Kismet to analyze wireless networks and capture wireless network traffic:**

- Kismet is a wireless network analyzer and packet sniffer that can capture wireless network traffic and analyze wireless networks.
- Knowledge of how to use Kismet to analyze wireless networks and capture wireless network traffic.

**Identify and deal with rogue router advertisements and DHCP messages:**

- Knowledge of how to identify and deal with rogue router advertisements and DHCP messages, which can be used by attackers to perform man-in-the-middle attacks.

**Awareness of aircrack-ng and bettercap:**

- aircrack-ng is a suite of tools that can be used to perform wireless network auditing and penetration testing, including cracking WPA and WPA2 keys.
- bettercap is a network attack tool that can be used to perform man-in-the-middle attacks and perform network reconnaissance.

The following is a partial list of the used files, terms and utilities:

- **radiusd:** The FreeRADIUS server daemon.
- **radmin:** A utility for managing the FreeRADIUS server.
- **radtest:** A utility for testing RADIUS authentication.
- **radclient**: A utility for sending RADIUS packets.
- **radlast:** A utility for showing the last logins of RADIUS users.
- **radwho**: A utility for showing who is currently logged in via RADIUS.
- **radiusd.conf:** The configuration file for the FreeRADIUS server.
- **/etc/raddb/***: The directory containing the FreeRADIUS server configuration files.
- **tshark:** A command-line interface to Wireshark.
- **tcpdump:** A command-line network packet analyzer.
- **kismet**: A wireless network analyzer and packet sniffer.
- **ndpmon:** A utility for monitoring and debugging IPv6 Neighbor Discovery Protocol (NDP) packets.

## 334.2 Network Intrusion Detection (weight: 4)

- Implement bandwidth usage monitoring
- Configure and use Snort, including rule management
- Configure and use OpenVAS, including NASL

**Implement bandwidth usage monitoring:**

- Bandwidth usage monitoring involves monitoring the traffic on a network and identifying the top users, applications, and protocols that are consuming the most bandwidth.
- **ntop** is a tool that can be used for bandwidth usage monitoring. It provides real-time network traffic analysis and can be used to generate reports on network usage.

## Configure and use Snort, including rule management:

- Snort is an open source intrusion detection and prevention system (IDS/IPS) that can be used to detect and prevent network attacks.
- The snort package includes the Snort IDS engine, as well as other utilities such as snort-stat for generating statistics on Snort alerts.
- **pulledpork.pl** is a Perl script that can be used to download and update Snort rules.
- **/etc/snort/** is the directory where Snort configuration files are stored.

## Configure and use OpenVAS, including NASL:

- OpenVAS is an open source vulnerability scanner that can be used to identify security vulnerabilities on a network.
- **openvas-adduser** and **openvas-rmuser** are utilities for adding and removing users to the OpenVAS system.
- **openvas-nvt-sync** is a utility for synchronizing the OpenVAS vulnerability database with the latest updates.
- **openvassd** is the OpenVAS scanner daemon that performs vulnerability scans.
- **openvas-mkcert** is a utility for creating SSL certificates for OpenVAS.
- **openvas-feed-update** is a utility for updating the OpenVAS vulnerability feeds.
- **/etc/openvas/** is the directory where OpenVAS configuration files are stored. NASL scripts can be stored in this directory and used to write custom vulnerability checks.

## 334.3 Packet Filtering (weight: 5)

- Understand common firewall architectures, including DMZ
- Understand and use iptables and ip6tables, including standard modules, tests and targets
- Implement packet filtering for IPv4 and IPv6
- Implement connection tracking and network address translation
- Manage IP sets and use them in netfilter rules
- Awareness of nftables and nft
- Awareness of ebtables
- Awareness of conntrackd

### Understand common firewall architectures, including DMZ:

A firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules.
**A demilitarized zone (DMZ)** is a separate network that sits between an organization's internal network and the internet, and is used to provide a buffer zone for external-facing services.
Understand and use iptables and ip6tables, including standard modules, tests and targets:
**iptables and ip6tables** are Linux-based firewalls that provide packet filtering and network address translation functionality.
**iptables** is used for filtering IPv4 traffic, while ip6tables is used for filtering IPv6 traffic.
iptables and ip6tables make use of modules, tests, and targets to allow for fine-grained control of network traffic.
Implement packet filtering for IPv4 and IPv6:

Packet filtering involves selectively allowing or blocking network traffic based on criteria such as source and destination IP addresses, ports, and protocols.
**iptables and ip6tables** can be used to implement packet filtering for IPv4 and IPv6 traffic, respectively.
Implement connection tracking and network address translation:

**Connection tracking** is a feature of iptables and ip6tables that tracks network connections and can be used to allow or block traffic based on the state of the connection.
**Network address translation (NAT)** is a feature that allows private IP addresses to be translated to public IP addresses, and vice versa. This can be used to provide internet access to devices on a private network.
Manage IP sets and use them in netfilter rules:

- IP sets are a way of grouping together IP addresses, ports, and protocols to simplify firewall rules.
- iptables and ip6tables support the use of IP sets in rules to allow for more efficient and flexible filtering.

**Awareness of nftables and nft:**

**nftables** is a newer firewall implementation that is designed to replace iptables.
nft is a tool for configuring and managing nftables.
Awareness of ebtables:

**ebtables** is a Linux-based firewall that provides filtering for Ethernet frames.
Awareness of conntrackd:

**conntrackd** is a daemon that can be used to synchronize connection tracking information between multiple firewalls.

**iptables and ip6tables** are command-line tools used to manage firewall rules on Linux systems. iptables handles IPv4 traffic, while ip6tables is used for IPv6 traffic. Both tools allow the administrator to define rules that determine what traffic is allowed or denied based on various criteria such as the source and destination IP address, port numbers, and protocol type.

**iptables-save and ip6tables-save** are used to save the current firewall rules in a format that can be easily restored at a later time. Conversely, iptables-restore and ip6tables-restore are used to load previously saved firewall rules.

**ipset** is a command-line utility used to manage sets of IP addresses or networks. It can be used in conjunction with iptables or ip6tables to simplify the management of firewall rules. For example, rather than defining multiple individual rules to allow or deny traffic from multiple IP addresses, an administrator could define an IP set and then reference it in a single rule.

**Here's an example of how to use ipset to manage a set of IP addresses and incorporate it into an iptables rule:**

Create an IP set named allowed_ips:

**sudo ipset create allowed_ips hash:ip**
This creates an empty IP set using the hash:ip type, which allows you to add individual IP addresses to the set.

Add some IP addresses to the set:

**sudo ipset add allowed_ips 192.0.2.1**
**sudo ipset add allowed_ips 192.0.2.2**
**sudo ipset add allowed_ips 192.0.2.3**
This adds three IP addresses to the allowed_ips set.
Create an iptables rule that references the allowed_ips set:
**sudo iptables -A INPUT -m set --match-set allowed_ips src -j ACCEPT**
This rule allows incoming traffic from any IP address that is included in the allowed_ips set.
Now, any incoming traffic from an IP address that is not in the allowed_ips set will be blocked by default, unless other iptables rules allow it. The benefit of using an IP set is that it allows you to easily manage and reference a large number of IP addresses in a single rule.
After you have created and configured your iptables rules, you can save them to a file using the **iptables-save or ip6tables-save command**, depending on whether you are working with IPv4 or IPv6 rules.
Here's an example of how to save your IPv4 iptables rules to a file:
**sudo iptables-save > /etc/iptables/rules.v4**

This command redirects the output of iptables-save to a file named rules.v4 in the /etc/iptables directory. By convention, this is where the Ubuntu distribution stores the iptables rules.

You can then restore the saved rules at boot time by adding the following line to the /etc/rc.local file:

**/sbin/iptables-restore < /etc/iptables/rules.v4**

This command reads the saved rules from the rules.v4 file and applies them to the system's iptables configuration. Alternatively, you can also load the saved rules manually using the iptables-restore command:

**sudo iptables-restore < /etc/iptables/rules.v4**

## 334.4 Virtual Private Networks (weight: 4)

- Understand the principles of bridged and routed VPNs
- Understand the principles and major differences of the OpenVPN, IPsec, IKEv2 and WireGuard protocols
- Configure and operate OpenVPN servers and clients
- Configure and operate IPsec servers and clients using strongSwan
- Configure and operate WireGuard servers and clients
- Awareness of L2TP

The key knowledge areas listed refer to Virtual Private Networks (VPNs) and their implementation. VPNs are used to establish a secure communication channel between two or more networks over the internet or other public networks. The areas covered include:

**Understand the principles of bridged and routed VPNs:** This involves understanding how VPNs can be implemented using different networking technologies such as bridged and routed networks.

**Understand the principles and major differences of the OpenVPN, IPsec, IKEv2, and WireGuard protocols: This involves understanding the different VPN protocols available, their strengths and weaknesses, and their use cases.**

OpenVPN, IPsec, IKEv2, and WireGuard are all popular VPN protocols that offer different features and advantages. Here are the principles and major differences of each:

**OpenVPN**

OpenVPN is a popular open-source VPN protocol that uses OpenSSL to create an encrypted connection. It can be used on a variety of platforms and can be configured to use a variety of encryption and authentication methods. OpenVPN is known for being highly configurable, flexible, and easy to use.

The major principles of OpenVPN are:

- Encryption: OpenVPN uses a combination of symmetric-key and public-key encryption to create an encrypted tunnel between two endpoints.
- Authentication: OpenVPN can use a variety of authentication methods, including username/password, public-key, and TLS certificates.
- Flexibility: OpenVPN is highly configurable and can be used in a variety of network topologies, including remote access, site-to-site, and client-server configurations.

**IPsec**

IPsec (Internet Protocol Security) is a protocol suite that provides security services for IP networks. It can be used to create VPNs, as well as to secure individual network connections. IPsec can be used with a variety of encryption and authentication methods, and is often used in enterprise networks.

The major principles of IPsec are:

- Encryption: IPsec can use a variety of encryption algorithms, including DES, 3DES, AES, and Blowfish.
- Authentication: IPsec can use a variety of authentication methods, including pre-shared keys, public-key, and digital certificates.
- Compatibility: IPsec is widely supported by most operating systems and network devices.

**IKEv2**

IKEv2 (Internet Key Exchange version 2) is a protocol used to create VPNs, often in combination with IPsec. It is known for being fast and efficient, and for its ability to quickly re-establish connections in the event of network disruptions. IKEv2 is often used in mobile and remote access scenarios.

The major principles of IKEv2 are:

- Encryption: IKEv2 can use a variety of encryption algorithms, including AES and 3DES.
- Authentication: IKEv2 can use a variety of authentication methods, including pre-shared keys, digital certificates, and Extensible Authentication Protocol (EAP).
- Efficiency: IKEv2 is designed to be fast and efficient, with minimal overhead.

**WireGuard**

WireGuard is a newer VPN protocol that aims to provide better performance and simpler configuration than other VPN protocols. It is known for its speed, simplicity, and minimal code base. WireGuard is often used in scenarios where performance and ease of use are a priority.

The major principles of WireGuard are:

- Encryption: WireGuard uses a combination of symmetric-key and public-key encryption to create an encrypted tunnel between two endpoints.
- Authentication: WireGuard uses public-key authentication to authenticate endpoints.
- Simplicity: WireGuard is designed to be simple and easy to configure, with a minimal code base.

Overall, the choice of VPN protocol depends on the specific use case and requirements. OpenVPN is often a good choice for its flexibility, while IPsec is a good choice for enterprise networks. IKEv2 is often used in mobile and remote access scenarios, while WireGuard is often used for its simplicity and performance.

**To configure and operate OpenVPN servers and clients, you can follow these steps:**

Install OpenVPN on your server and clients using the appropriate package manager for your operating system.

Generate cryptographic keys and certificates for the OpenVPN server and clients using the **easy-rsa tool** that comes with OpenVPN.

Configure the OpenVPN server by creating a configuration file in the /etc/openvpn/ directory. This file should specify the cryptographic keys and certificates, the network settings, and any other options you want to enable.

Start the OpenVPN server using the openvpn command and specifying the path to the configuration file you created in the previous step.

Configure the OpenVPN clients by creating a configuration file in the /etc/openvpn/ directory on each client. This file should specify the cryptographic keys and certificates, the server's IP address and port, and any other options you want to enable.

Start the OpenVPN clients using the openvpn command and specifying the path to the configuration file you created in the previous step.

**Here is an example configuration file for an OpenVPN server:**

# /etc/openvpn/server.conf

```
port 1194

proto udp

dev tun

ca /etc/openvpn/easy-rsa/keys/ca.crt

cert /etc/openvpn/easy-rsa/keys/server.crt

key /etc/openvpn/easy-rsa/keys/server.key
```

```
dh /etc/openvpn/easy-rsa/keys/dh.pem

server 10.8.0.0 255.255.255.0

ifconfig-pool-persist ipp.txt

push "route 192.168.0.0 255.255.255.0"

push "dhcp-option DNS 8.8.8.8"

keepalive 10 120

comp-lzo

user nobody

group nobody

persist-key

persist-tun

status openvpn-status.log

verb 3
```

This configuration file specifies that the OpenVPN server should listen on UDP port 1194 and use the TUN device. It also specifies the cryptographic keys and certificates, the IP address range for the VPN network, and various other options.

To start the OpenVPN server, you can use the following command:

```
# openvpn /etc/openvpn/server.conf
```

You can then create configuration files for the OpenVPN clients, specifying the server's IP address and port, and start the clients using the openvpn command

**Configure and operate IPsec servers and clients using strongSwan:** This involves configuring and managing IPsec servers and clients using the strongSwan open-source software, which is a widely used VPN protocol that provides strong security.

In this example, we will configure strongSwan to create an IPsec VPN between two servers running Ubuntu 20.04.

**Step 1: Install strongSwan on both servers**

Install strongSwan on both servers using the following command:

**sudo apt-get update**

**sudo apt-get install strongswan**

**Step 2: Configure the IPsec server**

On the first server, edit the /etc/ipsec.conf file to configure the IPsec server. Here's an example configuration:

# /etc/ipsec.conf

```
config setup

    charondebug="ike 2, knl 2, cfg 2"


conn %default

    ikelifetime=60m

    keylife=20m

    rekeymargin=3m

    keyingtries=1

    keyexchange=ikev2

    authby=secret
```

```
conn myvpn
    left=192.168.1.1
    leftsubnet=10.0.0.0/24
    leftid=@server1.example.com
    right=192.168.2.1
    rightsubnet=10.0.1.0/24
    rightid=@server2.example.com
    auto=add
```

The left parameter specifies the IP address of the local server, while the right parameter specifies the IP address of the remote server. The leftsubnet and rightsubnet parameters specify the subnets that are accessible on each side of the VPN. The leftid and rightid parameters specify the identity of each server in the VPN.

In this example, we're using IKEv2 for the key exchange and authenticating with a pre-shared key, specified in the /etc/ipsec.secrets file:

```
# /etc/ipsec.secrets
```

**@server1.example.com @server2.example.com : PSK "mysecretkey"**

Step 3: Configure the IPsec client

On the second server, edit the /etc/ipsec.conf file to configure the IPsec client. Here's an example configuration:

```
# /etc/ipsec.conf
```

```
config setup
    charondebug="ike 2, knl 2, cfg 2"

conn %default
    ikelifetime=60m
    keylife=20m
    rekeymargin=3m
    keyingtries=1
    keyexchange=ikev2
    authby=secret

conn myvpn
    left=192.168.2.1
    leftsubnet=10.0.1.0/24
    leftid=@server2.example.com
    right=192.168.1.1
    rightsubnet=10.0.0.0/24
    rightid=@server1.example.com
    auto=start
```

The configuration is similar to the server configuration, except that the left and right parameters are reversed, and the auto parameter is set to start the VPN automatically.

The pre-shared key is specified in the /etc/ipsec.secrets file, just like on the server:

# /etc/ipsec.secrets

**@server2.example.com @server1.example.com : PSK "mysecretkey"**

Step 4: Start the IPsec VPN

On both servers, start the IP

**Configure and operate WireGuard servers and clients:** This involves configuring and managing WireGuard servers and clients, which is a new and lightweight VPN protocol that is becoming increasingly popular.

WireGuard is a fast and modern VPN protocol that aims to provide better performance and simpler configuration than other VPN protocols. In this example, we will configure WireGuard to create a VPN between two servers running Ubuntu 20.04.

**Step 1: Install WireGuard on both servers**

Install WireGuard on both servers using the following command:

**sudo apt-get update**

**sudo apt-get install wireguard**

Step 2: Configure the WireGuard server

On the first server, create a configuration file for WireGuard in /etc/wireguard/wg0.conf:

```
[Interface]

Address = 10.0.0.1/24

PrivateKey = <server1-private-key>

ListenPort = 51820
```

```
[Peer]

PublicKey = <server2-public-key>

AllowedIPs = 10.0.0.2/32
```

In this configuration, we've set the server's IP address to 10.0.0.1/24 and specified a private key for the server. We've also specified the port that the server will listen on for incoming connections.

The [Peer] section specifies the configuration for the remote server. We've specified the public key of the remote server and allowed its IP address (10.0.0.2/32) to access the VPN.

Generate a private key for the server using the following command:

**sudo wg genkey | tee /etc/wireguard/privatekey | wg pubkey > /etc/wireguard/publickey**

Replace <server1-private-key> in the wg0.conf file with the private key generated in this step.

**Step 3: Configure the WireGuard client**

On the second server, create a configuration file for WireGuard in /etc/wireguard/wg0.conf:

```
[Interface]

Address = 10.0.0.2/24

PrivateKey = <server2-private-key>

[Peer]

PublicKey = <server1-public-key>

Endpoint = <server1-public-ip>:51820

AllowedIPs = 10.0.0.1/32
```

In this configuration, we've set the client's IP address to 10.0.0.2/24 and specified a private key for the client.

The [Peer] section specifies the configuration for the remote server. We've specified the public key of the remote server, its public IP address, and allowed its IP address (10.0.0.1/32) to access the VPN.

Generate a private key for the client using the following command:

**sudo wg genkey | tee /etc/wireguard/privatekey | wg pubkey > /etc/wireguard/publickey**

Replace <server2-private-key> in the wg0.conf file with the private key generated in this step.

**Step 4: Start the WireGuard VPN**

On both servers, start the WireGuard VPN using the following command:

**sudo wg-quick up wg0**

This will start the VPN and create the necessary network interface.

To stop the VPN, use the following command:

**sudo wg-quick down wg0**

That's it! Your WireGuard VPN is now configured and running between the two servers. You can test the VPN by pinging the other server's IP address from each server.

**Awareness of L2TP:** Layer 2 Tunneling Protocol (L2TP) is a protocol used to tunnel data packets between two locations in a VPN. Awareness of this protocol involves understanding how it works and its limitations.

The following is a partial lif used files, terms, and utilities are used in implementing VPNs:

**/etc/openvpn/:** Configuration files and scripts for OpenVPN

**openvpn:** Command-line tool to manage OpenVPN connections

**/etc/strongswan.conf**: Configuration file for strongSwan IPsec VPN software

**/etc/strongswan.d/**: Directory containing additional configuration files for strongSwan

**/etc/swanctl/swanctl.conf:** Configuration file for strongSwan swanctl utility

**/etc/swanctl/:** Directory containing additional configuration files for strongSwan swanctl utility

**swanctl:** Command-line utility for strongSwan configuration and management

**/etc/wireguard/**: Configuration files and scripts for WireGuard VPN

**wg**: Command-line tool to manage WireGuard connections

**wg-quick**: Script to quickly bring up or take down WireGuard connections

**ip**: Command-line tool to manage network interfaces and routes.

# *Topic 335: Threats and Vulnerability Assessment*

## 335.1 Common Security Vulnerabilities and Threats (weight: 2)

**Threats against individual nodes:**

- **Malware infections:** Malware is a type of software designed to cause harm to a computer system. It can infect individual nodes, such as laptops or servers, and spread to other systems within a network.
- **Physical theft or tampering:** Physical threats to individual nodes can include theft or tampering, which can compromise the confidentiality and integrity of sensitive data stored on the device.
- **Unauthorized access**: Unauthorized access to individual nodes can occur through hacking or exploitation of vulnerabilities, leading to unauthorized data access or theft.
- **Outdated software vulnerabilities:** Outdated software can contain known vulnerabilities that are easy targets for attackers.

**Threats against networks:**

- **DDoS attacks:** A Distributed Denial of Service (DDoS) attack is a type of attack that floods a network with traffic, overwhelming its resources and making it unavailable.
- **Man-in-the-middle attacks:** A man-in-the-middle (MitM) attack is a type of cyber attack where an attacker intercepts and alters communications between two parties, often without either party being aware.
- **Rogue devices/attacks from inside the network:** Rogue devices and attacks from within a network can be a major threat, as insiders may have access to sensitive data and systems.

- **Unsecured network protocols:** Unsecured network protocols can be vulnerable to exploitation by attackers, leading to data theft or unauthorized access.

**Threats against applications:**

- **SQL injections:** SQL injection is a type of attack where an attacker inserts malicious code into a web application's SQL statement, allowing them to gain unauthorized access to sensitive data stored in a database.
- **Cross-site scripting (XSS):** Cross-site scripting (XSS) is a type of attack that allows an attacker to inject malicious code into a web application, which can then be executed by unsuspecting users.
- **Remote code execution:** Remote code execution is a type of vulnerability that allows an attacker to execute code on a target system, often with the same privileges as the user running the vulnerable application.
- **Improper authentication and authorization:** Improper authentication and authorization can lead to unauthorized access to sensitive data and systems, as well as data theft.

**Threats against credentials and confidentiality:**

- **Phishing attacks:** Phishing is a type of social engineering attack where an attacker pretends to be a trusted entity in order to trick a user into revealing sensitive information, such as login credentials.
- **Social engineering:** Social engineering is the use of psychological manipulation to trick individuals into divulging sensitive information or performing actions that compromise their security.
- **Password cracking:** Password cracking is the process of attempting to guess or uncover a password, often using automated tools or exploiting vulnerabilities in password storage systems.

- **Unsecured data storage:** Unsecured data storage refers to the failure to properly secure sensitive data, such as through encryption or proper access controls, leaving it vulnerable to theft or unauthorized access.

**Honeypots:**

- **Decoy systems to distract attackers:** A honeypot is a decoy system designed to distract and divert attackers away from more valuable systems and data.
- **Collect information on attacker behavior and tools:** Honeypots can be used to collect information on attacker behavior and tools, providing valuable intelligence for security teams.
- **Help identify and mitigate emerging threats:** By monitoring attacker behavior, honeypots can help identify and mitigate emerging threats, improving overall security posture.
- **Monitor and detect unauthorized access attempts:** Honeypots can also be used to detect unauthorized access attempts, providing early warning of security incidents and helping to prevent more serious breaches.

**The following is a partial list of the used files, terms and utilities:**

**Trojans:** A Trojan is a type of malicious software that is disguised as a legitimate program. Trojans can be used to perform a variety of malicious activities, such as stealing sensitive information, installing other malicious software, or giving an attacker remote access to the infected system.

**Viruses:** A virus is a type of malicious software that replicates itself by infecting other files on a computer. Viruses can cause damage to files and systems, or use the infected computer as a platform for further attacks.

**Rootkits:** A rootkit is a type of malicious software that is designed to hide its presence and actions on an infected system. Rootkits can be used to gain persistent access to a system, bypass security measures, or steal sensitive information.

**Keylogger:** A keylogger is a type of malicious software or hardware device that records every keystroke made on a computer or device. Keyloggers are often used to steal passwords, credit card numbers, and other sensitive information.

**DoS and DDoS:** A Denial of Service (DoS) attack is an attempt to make a network resource unavailable to its intended users. A Distributed Denial of Service (DDoS) attack is a type of DoS attack that involves multiple devices attacking a single target. DoS and DDoS attacks can be used to disrupt online services, websites, and other networked systems.

**Man-in-the-Middle:** A man-in-the-middle (MitM) attack is a type of security attack where the attacker intercepts and alters the communication between two parties. MitM attacks can be used to steal sensitive information, alter data, or perform other malicious activities.

**ARP and NDP Forgery:** Address Resolution Protocol (ARP) and Neighbor Discovery Protocol (NDP) forgery are types of security attacks that involve falsifying ARP or NDP packets in a network. These attacks can be used to redirect network traffic, steal sensitive information, or perform other malicious activities.

**Rogue Access Points, Routers, and DHCP Servers:** Rogue access points, routers, and DHCP servers are unauthorized network devices that are placed on a network without the knowledge or consent of network administrators. These devices can be used to perform various malicious activities, such as intercepting network traffic, redirecting network traffic, or stealing sensitive information.

**Link Layer and IP Address Spoofing:** Link layer address spoofing and IP address spoofing are types of security attacks that involve falsifying the source address of a network packet. These attacks can be used to perform various malicious activities, such as intercepting network traffic, redirecting network traffic, or stealing sensitive information.

**Buffer Overflows:** A buffer overflow is a type of security vulnerability that occurs when a program writes more data to a buffer than it can hold. This can result in overwriting adjacent memory and causing the program to crash or execute arbitrary code.

**Cross-Site Request Forgery (CSRF):** Cross-site request forgery (CSRF) is a type of security Privilege Escalation: Privilege escalation is a type of security attack that involves a malicious actor gaining access to higher-level privileges on a computer system. This can be achieved through exploiting vulnerabilities in software, stealing credentials, or using social engineering tactics.

**Brute Force Attacks:** A brute force attack is a type of security attack that involves attempting to guess the password or key for a system or service by trying every possible combination. Brute force attacks can be used to gain unauthorized access to systems, steal sensitive information, or perform other malicious activities.
**Rainbow Tables:** A rainbow table is a precomputed table of hashes used in password cracking. Rainbow tables are used to speed up the process of guessing a password by reducing the number of hashes that need to be calculated.

## 335.2 Penetration Testing (weight: 3)

**Penetration testing, also known as ethical hacking**, is a simulated attack on a computer system or network to identify vulnerabilities and assess the security posture of the target. It is performed by ethical hackers who use the same tools and techniques as malicious hackers, but with the permission of the target organization.

**The legal implications of penetration testing** vary by jurisdiction and must be considered before conducting a test. In some countries, unauthorized access to computer systems is illegal, and penetration testers must have written permission from the target organization to perform the test.

**Penetration testing** is a method of evaluating the security of a computer system or network by simulating an attack. It involves several phases that help to identify vulnerabilities and evaluate the effectiveness of security measures. Here's an overview of the phases of a penetration test:

- **Active and passive information gathering:** This phase involves collecting information about the target system or network, both passively (without interacting with the target) and actively (by interacting with the target). Passive information-gathering techniques include researching public information about the target, while active techniques include using tools like **Nmap** to scan the target network.

- **Enumeration:** This phase involves actively interacting with the target to gather information about its systems, services, and users. This information can be used to identify potential vulnerabilities and attack vectors.
- **Gaining access:** This phase involves attempting to exploit vulnerabilities and gain access to the target system or network. This can involve techniques like brute force attacks, exploiting software vulnerabilities, or social engineering attacks.
- **Privilege escalation:** This phase involves increasing the level of access or control on the target system once access has been gained. This can be done by exploiting vulnerabilities in the operating system or applications, or by compromising additional systems or user accounts.

- **Access maintenance:** This phase involves maintaining access to the target system, including covering tracks to conceal the presence of the attacker and creating backdoors to allow future access.
- **Covering tracks:** This phase involves removing the evidence of the attack, such as log files or configuration changes, and cleaning up any malware or other malicious software that may have been installed during the attack.

These phases are not always performed in a strict order, and the focus and scope of a penetration test can vary depending on the specific requirements and objectives of the test. However, they provide a general framework for conducting a penetration test and evaluating the security of a computer system or network.

**Metasploit** is a popular framework for security testing that includes a collection of modules for various purposes. These modules can be divided into three main categories: exploits, payloads, and auxiliary modules.

- **Exploits:** Exploits are modules that take advantage of vulnerabilities in software or systems to gain unauthorized access or control. Exploits are typically used to deliver payloads to target systems.
- **Payloads:** Payloads are modules that are executed on the target system once an exploit has been successful. Payloads can perform a variety of actions, such as capturing keystrokes, stealing data, or establishing a reverse shell connection back to the attacker.
- **Auxiliary modules:** Auxiliary modules are modules that perform a variety of other functions, such as reconnaissance, network scanning, and vulnerability analysis. Auxiliary modules can be used to gather information about a target system or network, identify potential vulnerabilities, and assist in the execution of exploits and payloads.

Each of these types of modules has a specific purpose and can be used together to achieve different objectives during a penetration test. The Metasploit framework provides a flexible and modular platform for security testing that can be customized to meet specific needs.

**Nmap** is a tool used for network exploration and security auditing. It allows users to scan networks and hosts to identify open ports, operating systems, and running services. Nmap supports different scan methods, including version scans and operating system recognition, and also has a scripting engine that allows users to automate and customize their scans. Here are some of the most important Nmap scans and the corresponding commands:

- **Ping scan (-sP):** This scan is used to determine if a host is up and responding. It sends a simple ping request to the target host and reports if it received a response.
  Command: **nmap -sP <target_host>**
- **Port scan (-p):** This scan is used to identify open ports on a target host. It sends packets to the specified ports and analyzes the response to determine if the port is open, closed, or filtered.
  Command: **nmap -p <port_range> <target_host>**
- **Version scan (-sV):** This scan is used to determine the version of services running on a target host's open ports. It sends requests to the open ports and analyzes the responses to determine the software and version being used.
  Command: **nmap -sV <target_host>**
- **OS scan (-O):** This scan is used to determine the operating system running on a target host. It analyzes the responses from the target host to determine the operating system type and version.
  Command: **nmap -O <target_host>**
- **Stealth scan (-sS):** This scan is a type of port scan that is designed to be less detectable. It uses a technique called half-open scanning, where only a single packet is sent to the target host to determine if the port is open.
  Command: **nmap -sS <target_host>**

- **TCP scan:** A TCP scan is used to determine if a target host's TCP ports are open, closed, or filtered. It sends packets to the target host's TCP ports and analyzes the responses to determine the status of the port.
  Command: **nmap -sT <target_host>**

- **UDP scan:** A UDP scan is used to determine if a target host's UDP ports are open, closed, or filtered. It sends packets to the target host's UDP ports and analyzes the responses to determine the status of the port.
  Command: **nmap -sU <target_host>**

It is important to note that UDP scans are typically slower than TCP scans because the UDP protocol does not provide the same level of reliability as TCP. As a result, UDP scans may not receive a response from all target hosts, and false negatives can occur. However, UDP scans are useful for identifying open UDP ports, which can sometimes be overlooked during a TCP scan.

**Kali Linux** is a widely used distribution of Linux designed specifically for security testing and ethical hacking. It comes pre-installed with a variety of tools, including Nmap, Metasploit, and Armitage, a graphical interface for Metasploit. **The Social Engineer Toolkit (SET)** is another tool that can be used for penetration testing and is focused on exploiting human vulnerabilities through social engineering techniques.