

# Appunti di Internet Security - Ramaci Emanuel

## Introduzione

Oggi l'informatica è ovunque: dalla spesa al supermercato al prelievo al bancomat, ogni nostra azione quotidiana si appoggia su sistemi digitali. Ma più questi sistemi diventano pervasivi, più cresce il bisogno di proteggerli.

**La sicurezza informatica non è un optional**, ma una componente fondamentale per garantire che tutto ciò che di digitale utilizziamo — dati, identità, transazioni — resti integro e riservato.

Prendiamo il caso del bancomat: se il sistema non fosse sicuro, chiunque potrebbe intercettare le credenziali o le transazioni, con conseguenze gravi per tutti.

Per evitare scenari simili, è essenziale **capire a fondo il sistema che vogliamo proteggere**, perché solo conoscendolo possiamo individuarne le **potenziali vulnerabilità** e intervenire in modo mirato.

## Terminologia

Quando si parla di **sicurezza informatica**, è facile fare confusione tra termini che sembrano simili ma hanno **significati molto diversi**. Capirli bene fin da subito è fondamentale per evitare fraintendimenti più avanti.

### Safety

- **Definizione:** riguarda la **protezione delle persone** (fisica e mentale), non dei dati o dei sistemi.
- **Esempio:** un sistema industriale che evita esplosioni o ferimenti garantisce *safety*.
- **Estendibile anche agli ambienti**, come luoghi di lavoro, strutture sanitarie, ecc.

### Security

- **Definizione:** è la **protezione dei sistemi, delle reti e delle informazioni** da minacce, attacchi o accessi non autorizzati.
- **Esempio:** un firewall o un protocollo crittografico come TLS servono a garantire *security*.
- **Focus:** attacchi, vulnerabilità, minacce informatiche.

### Privacy

- **Definizione:** è il **diritto** di ogni individuo a **controllare l'accesso e l'uso delle proprie informazioni personali**, sia nel mondo fisico che digitale.
- **Esempio:** poter decidere chi vede i tuoi dati medici o il tuo profilo social.
- **Non è un mezzo tecnico**, ma un concetto giuridico e sociale.

### Data Protection

- **Definizione:** è l'**insieme delle misure (tecniche e organizzative)** che **realizzano concretamente il diritto alla privacy**.
- **Esempio:** crittografia, backup sicuri, autenticazione a due fattori.
- **In sintesi:** la *privacy* è il diritto, la *data protection* è il mezzo per garantirla.

### Trust

- **Definizione:** è la **fiducia che un utente ripone** in un sistema, un servizio o un'organizzazione.
- **Esempio:** quando inserisci la tua carta di credito in un sito e-commerce, stai "fidandoti" che quel sito non farà un uso scorretto dei tuoi dati.
- **Nota:** non esiste un sistema *perfettamente sicuro*. Un certo grado di fiducia sarà **sempre necessario**, e in caso di problemi spesso intervengono **assicurazioni** per coprire eventuali danni.

### Sicurezza = processo in evoluzione

Questa terminologia ci fa subito capire una cosa importante:

**La sicurezza non è un risultato fisso, ma un processo continuo.**

- Si evolve nel tempo.
- Si adatta alle nuove tecnologie e minacce.

- Cerca sempre un **compromesso tra funzionalità e protezione**: un sistema troppo sicuro potrebbe essere inutilizzabile, uno troppo comodo potrebbe essere vulnerabile.

## Trojan

Il **Trojan** (abbreviazione di "Trojan Horse", cioè *cavallo di Troia*) è un **programma che sembra innocuo ma nasconde un codice malevolo**. Proprio come nel mito greco, dove i soldati erano nascosti dentro il cavallo, qui il codice dannoso è nascosto dentro qualcosa che sembra utile o sicuro (un PDF, un programma, un'app, ecc.).

Spesso i Trojan vengono diffusi tramite **ingegneria sociale**: l'utente viene convinto a **scaricare ed eseguire volontariamente** il programma, perché crede che sia legittimo.

### Digressione tecnica: ACL & SUID

Prima di entrare nel funzionamento di un esempio pratico, serve capire un concetto chiave dei sistemi Linux/Unix.

#### ACL (Access Control List)

Ogni file ha permessi associati, divisi in tre gruppi:

- **User owner**: chi possiede il file
- **Group**: il gruppo a cui appartiene l'utente
- **Others**: tutti gli altri

Per ogni gruppo c'è una **tripletta di permessi**: `r` (read), `w` (write), `x` (execute).

Ad esempio: `-rwxr-x--` → il proprietario può fare tutto, il gruppo può leggere/eseguire, gli altri nulla.

#### SUID (Set User ID)

È un bit speciale che, se attivo su un file eseguibile, **fa sì che quel programma venga eseguito con i permessi del proprietario**, non di chi lo avvia.

Esempio:

Il comando `passwd` permette a ogni utente di cambiare la propria password.

Anche se il file è di proprietà di **root**, viene eseguito con i suoi privilegi grazie al **bit SUID**.

### Il Trojan "ls" – Analisi passo passo

Immagina che qualcuno ti mandi un file chiamato `ls` (lo stesso nome del famoso comando per listare i file). Sembra innocente, no? Ma dentro c'è:

```
cp /bin/sh /tmp/.xxsh
chmod u+s,o+x /tmp/.xxsh
rm ./ls
ls $*
```

#### Cosa fa, in ordine:

1. `cp /bin/sh /tmp/.xxsh`  
Copia la shell `/bin/sh` in un file nascosto chiamato `.xxsh` nella cartella `/tmp`.  
(Cartella scrivibile da chiunque → già pericolosa)
2. `chmod u+s,o+x /tmp/.xxsh`  
Imposta il **bit SUID** sul nuovo file `.xxsh` e lo rende eseguibile da chiunque (`o+x`).  
Se la copia ha mantenuto `root` come owner → BOOM: shell con privilegi di root.
3. `rm ./ls`  
Cancella sé stesso. Così sparisce ogni traccia apparente.
4. `ls $*`  
Esegue il vero comando `ls` con gli stessi argomenti ricevuti.  
Così la vittima non si accorge di nulla: il comando "funziona" come sempre.

#### Cosa significa "stessi argomenti ricevuti"?

Quando esegui uno script nel terminale (come il nostro Trojan `ls`), puoi passargli degli argomenti.

Per esempio:

```
ls -la /etc
```

Qui stai dicendo: "esegui `ls` con i parametri `-la /etc`", cioè: lista dettagliata di `/etc`.

All'interno dello script, la variabile speciale `$*` contiene tutti gli argomenti che l'utente ha passato allo script.

#### Perché viene usato `ls $*`?

Perché lo script vuole simulare il comportamento del comando `ls` vero, così la vittima non si insospettisce.

Quindi:

- Se la vittima scrive semplicemente `ls`
  - `$*` è vuoto
  - il Trojan esegue `ls` normale → mostra la lista dei file nella cartella corrente
- Se scrive `ls -l /home/emanuel`
  - `$*` sarà `-l /home/emanuel`
  - il Trojan esegue: `ls -l /home/emanuel` → output normale

Così l'utente non si accorge che ha eseguito uno script infetto.

Dunque, la vera parte dell'attacco è:

Creare una shell (`/tmp/.xxsh`) con il bit SUID attivo e di proprietà root, accessibile a chiunque.

Questo vuol dire che in un secondo momento, anche un utente normale potrà fare:

```
/tmp/.xxsh
```

E puff! avrà una shell di root.

#### Come sarebbe un abuso?

Immagina che l'attaccante entri nella macchina come utente normale, poi esegue:

```
/tmp/.xxsh
```

e si ritrova con:

```
# whoami  
root
```

Missione compiuta.

**La vittima, per far sì che il Trojan venga eseguito, deve lanciare `./ls` o semplicemente `ls`?**

#### Perché funzionava?

Fino a qualche anno fa, molti sistemi Linux avevano nel `PATH` la directory `.` (cioè la cartella corrente).

E spesso il punto veniva cercato per primo!

Quindi: se scarichi un file chiamato `ls` nella tua home e scrivi `ls` nel terminale, esegui per primo quello nella tua cartella, non il comando ufficiale.

**La vittima quindi esegue inconsapevolmente il Trojan.**

#### Perché funziona senza `./`?

Tutto dipende dalla variabile d'ambiente `PATH`.

Questa variabile contiene una lista di directory da cercare quando esegui un comando. Ad esempio, se fai:

ls

il sistema cerca ls in ognuna delle cartelle elencate in PATH, in ordine.

### Il trucco del passato:

Una volta (e purtroppo a volte ancora oggi in ambienti mal configurati), la directory corrente (.) era in cima alla variabile PATH.

Esempio:

```
PATH=.:./usr/bin:/bin:/usr/local/bin:....
```

Quindi, se nella tua cartella c'era un file eseguibile chiamato ls, e tu scrivevi:

ls

Il sistema prima guarda ./ls (quello presente nella tua cartella), e poi il vero /bin/ls.

### Risultato?

Se scarichi o ricevi (tramite social engineering) un file chiamato ls, e stai nella cartella in cui è presente, eseguendo ls pensi di usare il comando vero, ma in realtà avvii il Trojan!

### Come è stato mitigato oggi?

1. I file scaricati vanno in ~/Downloads e non sono eseguibili.
2. Il PATH non include più . o lo mette in fondo, mai all'inizio.
3. Alcuni antivirus avvertono se un file in /tmp ha il bit SUID.
4. Le moderne distro controllano meglio chi è l'owner dei file copiati (es. cp potrebbe cambiare l'owner → meno rischio).

### Difesa in profondità

Non si fa affidamento su una sola barriera, ma su più strati di sicurezza:

- Ambiente isolato per i download
- Permessi bloccati
- PATH sicuro
- Monitoraggio di file sospetti

Così, se una protezione viene aggirata... ce ne sono altre pronte.

### Strumenti di sicurezza

Abbiamo detto che anche un attacco banale come quello di un Trojan può costringerci a studiare a fondo il funzionamento del sistema operativo. E questo è solo uno dei tanti esempi che affronteremo. Ma quindi... come possiamo proteggerci dagli attacchi informatici? Quali strumenti abbiamo?

#### Cos'è la sicurezza?

La sicurezza non è qualcosa di statico o assoluto, ma un processo continuo.

Significa:

- Individuare gli anelli deboli in un sistema.
- Applicare contromisure mirate rispetto alle minacce che vogliamo contrastare.
- Bilanciare costi e benefici, perché mettere in sicurezza un sistema ha sempre un costo (tempo, risorse, prestazioni).
- Agire su più livelli: nessuna misura da sola è sufficiente, la sicurezza va costruita a strati.

### I rischi che minacciano un sistema

Un sistema può essere vulnerabile per diversi motivi, anche se non sembrano evidenti a prima vista:

#### 1. Complessità del sistema

- Più funzioni ha un sistema, più è **difficile proteggerlo completamente**.
- Maggiore complessità = più superfici d'attacco.

#### 2. Combinazione di sistemi

- Anche se due sistemi singolarmente sono sicuri, **la loro integrazione** potrebbe generare nuovi problemi o bug imprevisti.

#### 3. Bug e vulnerabilità

- Ogni software ha bug. Alcuni sono innocui, ma altri possono essere sfruttati da un attaccante come vere e proprie **vulnerabilità**.
  - **Bug**: è un **errore nel codice** o nel comportamento di un software. Può causare malfunzionamenti, ma **non sempre è pericoloso** per la sicurezza.
  - **Vulnerabilità**: è un **bug (o configurazione errata)** che può essere **sfruttato da un attaccante** per compromettere la sicurezza del sistema.
  - Tutte le vulnerabilità sono bug, **ma non tutti i bug sono vulnerabilità**.

#### 4. Proprietà emergenti

- Le nuove tecnologie (AI, blockchain, IoT...) portano **novità entusiasmanti**, ma anche **rischi mai visti prima**. Non sempre esistono contromisure già pronte.

#### 5. Fattore umano

- Puoi avere il miglior sistema di sicurezza del mondo, ma basta **una persona distratta o manipolata** per compromettere tutto (es. phishing, ingegneria sociale).

---

### Sicurezza = Attacco vs Difesa

La sicurezza può essere vista come una **continua battaglia tra attaccanti e difensori**.

Questa logica è incarnata nei concetti di:

- **Red Team**: simula gli attacchi, cerca falliche e prova a violare i sistemi.
- **Blue Team**: difende il sistema, applica misure di protezione, monitora e risponde agli attacchi.

---

### Strumenti fondamentali per la difesa

Vediamo ora gli strumenti principali che possiamo usare per mettere in sicurezza un sistema:

#### Crittografia (simmetrica e asimmetrica)

- **Serve a proteggere i dati** nascondendone il contenuto.
- **Simmetrica**: stessa chiave per cifrare e decifrare.
- **Asimmetrica**: coppia di chiavi pubblica/privata.
- **Nota bene**: anche se un algoritmo è sicuro *matematicamente*, la sua **implementazione software potrebbe avere bug**.

#### Policy (regole di sicurezza)

- Sono le **regole organizzative e tecniche** su:
  - Accesso ai dati
  - Protezione della privacy
  - Criteri per la gestione delle password
  - ...e tanto altro
- Serve per **definire chiaramente chi può fare cosa** e con quali limiti.

#### Autenticazione (conoscenza, possesso, biometria)

- **Qualcosa che sai**: password

- **Qualcosa che hai:** badge, token, smart card
- **Qualcosa che sei:** impronte digitali, riconoscimento facciale
- Serve applicare policy anche qui (es. lunghezza password, scadenza, numero di tentativi, ecc.)

#### Programmi di protezione

- Antivirus
- Firewall
- IDS/IPS (sistemi di rilevamento/prevenzione delle intrusioni)
- Mai dare per scontato che funzionino sempre perfettamente: **vanno aggiornati e testati.**

#### Protocolli di sicurezza

- **SSL/TLS, SSH, HTTPS, ecc.**
- Servono a **proteggere le comunicazioni in rete**, garantendo confidenzialità, integrità e autenticazione.

#### Formazione degli utenti

- Spesso l'**utente è il punto più debole.**
- Istruirlo serve a evitare errori banali ma pericolosissimi:
  - cliccare su un link sospetto
  - condividere una password
  - usare chiavette USB non controllate

#### Attacchi e attaccanti

Quando analizziamo un attacco informatico, dobbiamo cambiare prospettiva e pensare **con la testa dell'attaccante**. Questo significa non solo capire come funziona un sistema, ma anche **come può essere aggirato**.

Oltre ai concetti classici di rischio e vulnerabilità, è importante capire **le dinamiche moderne degli attacchi**, che sono cambiate tantissimo col tempo.

---

#### Caratteristiche dei rischi digitali moderni

Ecco quattro concetti chiave che ci fanno capire perché oggi **difendere un sistema è più difficile**:

##### 1. Automazione offensiva

- Gli attacchi possono essere **automatizzati**: non serve un hacker seduto davanti al PC a premere tasti.
- Esempio classico: bots che rubano **un centesimo** da ogni transazione VISA. Preso da solo è nulla, ma moltiplicato per milioni di carte = furto enorme.
- È il principio del "piccolo furto, ma su larga scala".

##### 2. Nessun limite geografico

- Un attacco può partire **da qualsiasi parte del mondo**.
- Chi ti attacca non ha bisogno di essere fisicamente vicino: **la rete collega tutto**.
- Questo complica molto l'attribuzione e la difesa.

##### 3. Condivisione di tecniche

- Oggi è **facile trovare strumenti di hacking** pronti all'uso su forum, Telegram, GitHub, ecc.
- Anche persone **senza grandi competenze** possono lanciare attacchi sfruttando un software già pronto.
- Si parla spesso di *script kiddie*, cioè chi usa strumenti fatti da altri.

##### 4. Difficoltà di reazione

- A volte, **non ci accorgiamo nemmeno di un attacco**.
- Esempio: il *port scanning*, nato per testare servizi, oggi viene usato per raccogliere info su un sistema da attaccare.
- Le azioni sospette possono sembrare normali se non si è vigili.

### **Tipi di attacco e contromisure (caso: furto fisico)**

Non tutti gli attacchi sono uguali: **cambiano per obiettivo e metodologia**. Vediamo un esempio molto concreto: **furto fisico di un dispositivo** (tipo laptop o smartphone).

#### **1. Attacco fondamentale: accesso al sistema**

- Obiettivo: accedere al dispositivo
- Contromisure: autenticazione (password, impronta, PIN)

#### **2. Attacco successivo 1: utilizzo delle funzionalità**

- Obiettivo: usare servizi o software presenti nel sistema
- Contromisure: autenticazioni a più livelli (es. per aprire un'app o accedere a dati specifici)

#### **3. Attacco successivo 2: accesso ai dati sensibili**

- Obiettivo: rubare password, file personali, ecc.
- Contromisure: crittografia dei dati o dell'intero filesystem

### **Ma nella realtà...**

Tutto questo **sulla carta funziona**. Ma nel mondo reale? Ecco i problemi comuni:

- L'autenticazione può essere **bypassata**: basta avviare il device con una USB live.
- Le password spesso sono **salvate nei browser**, senza protezione.
- La crittografia, per comodità, è **disabilitata**: così i dati sono leggibili se qualcuno accede fisicamente al device.

---

### **Altri tipi di attacco**

#### **Pirateria digitale**

- **Furto di identità**
- **Furto di proprietà intellettuale** (codice, design, musica, ecc.)
- **Furto di marchio**
- Impattano su persone e aziende, anche legalmente.

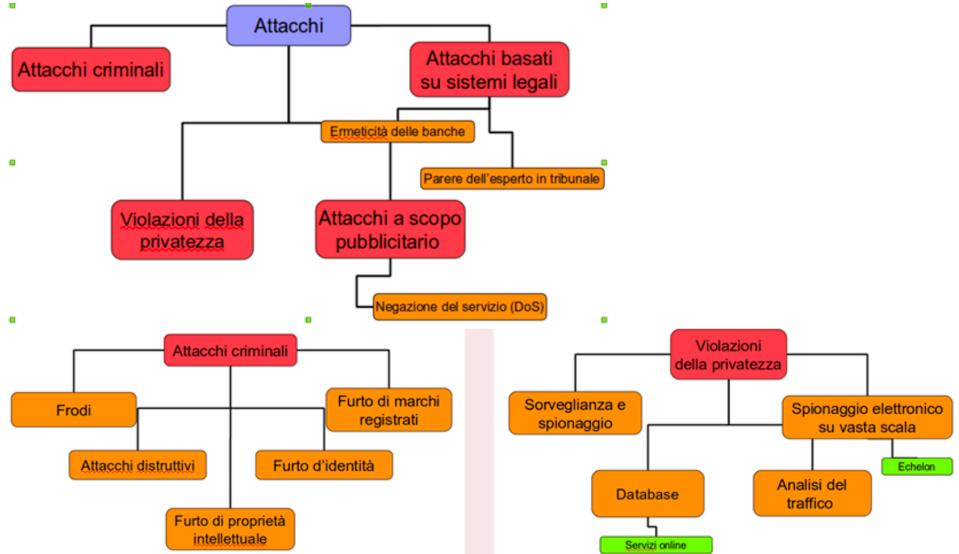
#### **Sniffing**

- Un attaccante può **intercettare il traffico** tra due dispositivi su una rete (es. Wi-Fi non protetta).
- È come se si piazzasse in mezzo e **spiasse tutto** quello che passa.
- Tecnica base di **attacchi man-in-the-middle**.

---

### **Classificare gli attacchi: in base all'obiettivo**

Ci sono tanti modi per classificare gli attacchi. Uno dei più interessanti è **in base all'obiettivo**.



Ad esempio:

- **Attacchi ai sistemi legali**
  - Sì, anche la legge può essere un bersaglio!
  - Esempio: **falsa testimonianza** in tribunale da parte di un consulente tecnico.
  - È un attacco che sfrutta il sistema legale, ma **non è etico** ed è **pericoloso**, perché distorce la giustizia.

## Proprietà di sicurezza

Quando si parla di **sicurezza informatica**, non si parla solo di bloccare attacchi. Si parla di **proteggere certe caratteristiche fondamentali delle informazioni**.

Queste caratteristiche sono chiamate **proprietà di sicurezza**.

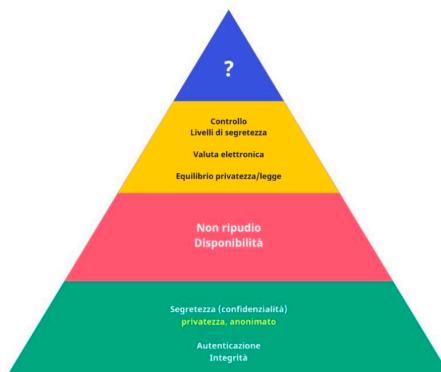
### Le 3 proprietà fondamentali (modello CIA)

Secondo lo **standard ISO 27001**, le tre proprietà base della sicurezza sono:

1. **Confidenzialità (Confidentiality)**
2. **Integrità (Integrity)**
3. **Disponibilità (Availability)**

In gergo: **CIA**

Tuttavia, **per il corso**, al posto della *Disponibilità* ci concentriamo di più su un'altra proprietà molto importante: **l'Autenticazione**.



## Segretezza (o confidenzialità)

Questa è la proprietà per cui **solo chi è autorizzato può accedere a certe informazioni**.

In pratica, un "segreto" crea due mondi:

- Chi **può sapere**
- Chi **non deve sapere**

Per proteggere la segretezza, entrano in gioco due concetti fondamentali:

- **Autenticazione** → per sapere *chi* è un utente
- **Policy di accesso** → per decidere *cosa può fare* quell'utente

### Tecniche per proteggere la segretezza

#### Crittografia

- Trasforma un messaggio in qualcosa di incomprensibile senza una chiave.
- Obiettivo: **rendere non deducibile** il contenuto del messaggio.

#### Steganografia

- Nasconde **il fatto stesso che esista un messaggio**.
- Esempio: si possono nascondere dati nel **least significant bit di un'immagine**.
- Obiettivo: **rendere non distinguibile** la presenza del messaggio.

Differenza:

- Crittografia: nasconde il *significato*
- Steganografia: nasconde *l'esistenza*

## Autenticazione

Questa proprietà serve a **verificare l'identità** di chi compie un'azione o accede a un sistema.

È fondamentale per creare **fiducia tra le parti**.

#### Tipi di autenticazione:

- **Basata su conoscenza** → es. password
- **Basata sul possesso** → es. smart card
- **Basata su caratteristiche biometriche** → es. impronta, volto

Nota importante:

- Le password si possono cambiare, le impronte no.
- Tutti i metodi hanno **vulnerabilità** (furti, cloni, ecc.)

## Integrità

Integrità significa che i **dati non sono stati modificati** da qualcuno che non era autorizzato.

Se io ti mando un file, voglio essere sicuro che arrivi esattamente come l'ho spedito, senza manomissioni.

#### Problema:

- I pacchetti di rete possono subire errori per **interferenze naturali**
- Ma oggi il vero problema è **l'attaccante intenzionale**, non l'errore casuale.

#### Soluzioni:

- **Checksum** → ok per errori naturali, **non basta** contro un attaccante.
- **Firma digitale** → molto più forte, perché:
  - Garantisce integrità
  - Dimostra anche l'identità di chi ha firmato

## Privatezza (Privacy)

Questa proprietà tutela **il diritto di una persona di scegliere se fornire o meno i propri dati**.

È una questione **di controllo personale**.

**Due livelli:**

- **Hard privacy** → tutela molto forte e rigida
- **Soft privacy** → più flessibile, spesso legata al consenso

Nell'UE, la privacy è **protetta per legge** (es. GDPR). Negli USA... molto meno.

## Anonimato

L'anonimato è la **privatezza dell'identità**: la possibilità di non rivelare *chi sei* durante una comunicazione.

Sembra parte della privacy, ma ha dinamiche e limiti diversi.

**Due tipi:**

- **Anonimato vero** → nessuno può risalire alla tua identità
- **Pseudo-anonimato** → sembri anonimo, ma qualcuno *potenzialmente* può scoprire chi sei

**Livelli di anonimato:**

### 1. Navigazione anonima

- Es. browser in incognito → **limitato**: l'IP è comunque tracciabile

### 2. Anonimato applicativo

- Es. **proxy server** → inoltra richieste al posto tuo, ma sa chi sei
- A differenza della VPN, **non cifra il traffico**

### 3. Anonimato a livello di routing

- Es. **Tor (The Onion Router)** → protegge l'identità con 3 nodi casuali:
  - Guard node: conosce il tuo IP, non la destinazione
  - Middle node: "spezza" il percorso
  - Exit node: vede la destinazione, ma **non sa chi sei**
- Ogni nodo decifra solo **uno strato** → come una cipolla

## Non ripudio

Il **non-ripudio** è la proprietà per cui **nessuno può negare di aver fatto qualcosa**.

| Se firmi digitalmente un documento, non puoi dire dopo che non sei stato tu.

**Collegamento:**

- Il non ripudio **richiede autenticazione**.
- Ma **autenticazione da sola non basta** per garantire non ripudio.

**Esempio pratico:**

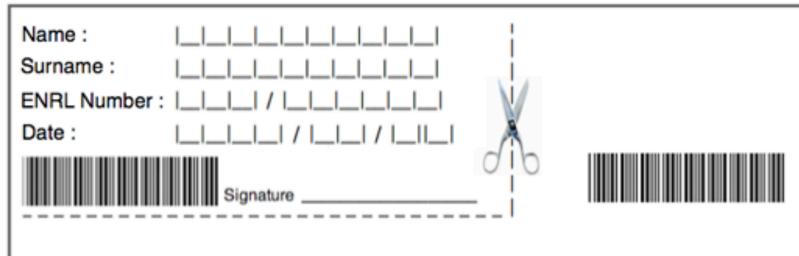
- Autenticarti con password su un sito non garantisce che **non possa negare in futuro** di essere stato tu.
- Una **firma digitale con certificato valido**, invece sì.

Nota finale:

Se hai **non ripudio, non puoi avere anonimato**. Sono due proprietà **opposte**.

## WATA (Written Authenticated Though Anonymous)

È un protocollo pensato per gli **esami ufficiali** (come i concorsi pubblici) in cui vogliamo **autenticare il candidato ma mantenere il compito anonimo**, per garantire **onestà e imparzialità**.



1) How Did He/She invent the hot water?

### Obiettivo del protocollo WATA

Garantire **due proprietà fondamentali** in un esame:

1. **Autenticazione** → Chi svolge il compito è davvero chi dice di essere.
2. **Anonimato** → Chi corregge il compito non deve sapere chi l'ha svolto, per evitare favoritismi.

### I ruoli coinvolti

- **Examiner** → Chi **prepara e corregge** il compito.
- **Invigilator** → Chi **sorveglia** durante l'esame (tipo il prof in aula).
- **Candidato** → Chi **sostiene** l'esame.

### Come funziona il protocollo (WATA2)

#### 1. Preparazione dell'esame (Examiner)

- L'examiner **estrae domande a caso** da un database.
  - Ogni test riceve un **ID randomico** → serve per tenere traccia del compito senza associare subito al candidato.
  - Si stampa il test con:
    - modulo di autenticazione (dove metti nome e firma)
    - barcode (lo stesso sia sulla parte dell'autenticazione sia sul compito)
    - domande e spazio per risposte
  - La parte importante: si fa una **firma "a cavallo"** tra il barcode e il modulo di autenticazione.
- Questa firma serve per **rendere difficile manomettere il foglio**.

**Il barcode di cui si parla è proprio quello stampato sul foglio delle domande e risposte** del test. La "firma a cavallo" significa che l'examiner mette una firma (o un timbro) che **"lega"** **insieme il barcode e il modulo di autenticazione** del candidato. In pratica, la firma attraversa entrambi, collegandoli in modo da renderne difficile la manomissione o la sostituzione separata. Quindi se qualcuno cercasse di modificare il foglio o separare il barcode dal modulo di autenticazione, la firma non combacerebbe più e si scoprirebbe subito la frode. È come una sorta di **sigillo di sicurezza fisico** per proteggere l'integrità del test e dell'identità del candidato!

#### 2. Distribuzione (Invigilator)

- L'invigilator **distribuisce a caso i test** ai candidati.
- Il candidato **compila il modulo di autenticazione** (cioè, si identifica).
- L'invigilator **verifica l'identità** con il documento.
- La parte delle domande/barcode **rimane coperta**, così l'invigilator **non sa che test ha preso chi** → questo preserva l'anonimato.

### **3. Svolgimento e consegna (Candidato + Invigilator)**

- Il candidato **compila il test**, poi **stacca il modulo di autenticazione**.
- Il compito (con domande + risposte) **viene consegnato separatamente** dal modulo identificativo.
- L'invigilator raccoglie tutto e lo dà all'examiner.

### **4. Correzione (Examiner)**

- L'examiner **corregge il compito** senza sapere di chi è → anonimato garantito.
- Assegna un **voto all'ID** del test.
- Solo **dopo**, quando il candidato torna per verbalizzare, si può **riassociare il modulo di autenticazione** al test grazie al barcode → così si sa a chi va il voto.

---

#### **Proprietà di sicurezza**

- Il sistema garantisce:
  - **Autenticazione certa** all'inizio
  - **Anonimato totale** durante la correzione
  - **Tracciabilità e integrità** grazie al barcode e alla firma a cavallo

---

#### **Punti deboli**

- **Non resiste alla collusione tra candidato ed examiner** (es. se si accordano per falsificare un voto).
- Anche se **invigilator ed examiner sono collusi**, **la struttura di WATA cerca comunque di garantire anonimato**.

#### **Collusione tra candidato ed examiner**

- **Collusione** significa che due o più persone **si mettono d'accordo** per fare qualcosa di illecito insieme.
- Questa alleanza rompe il principio di **anonimato e correttezza**, perché il candidato può essere identificato dall'examiner e ricevere un trattamento speciale.

---

#### **Collusione tra invigilator ed examiner**

- Qui la collusione è tra chi **sorveglia l'esame (invigilator)** e chi **corregge (examiner)**.
- Se invigilator ed examiner si accordano, per esempio per barare insieme, allora potrebbero scambiarsi informazioni su chi ha fatto cosa.
- Però, secondo il protocollo WATA, anche in questo caso, grazie a come è strutturato il sistema (firma a cavallo, barcode, separazione moduli), **l'anonimato è comunque protetto meglio** rispetto a sistemi tradizionali.
- Quindi, anche se collaborano per barare, è più difficile rompere l'anonimato rispetto alla collusione diretta con il candidato.

---

#### **La "doppia busta"? Superata!**

Il vecchio metodo della **doppia busta**:

- Compito dentro una busta
- Dati anagrafici dentro un'altra
- Examiner avrebbe dovuto correggere prima e associare poi...

Ma nulla impediva all'examiner di sbirciare prima i dati.

Quindi il **protocollo non era affidabile**, ed è qui che **WATA fa meglio**.

---

#### **WATA3 → Per gli esami in remoto**

Problema: manca il **token fisico** (cioè la parte cartacea).

Soluzione:

- Il candidato si **autentica digitalmente**
- Inserisce un **codice/token segreto** (che può essere compromesso, attenzione!)

- Il sistema **non associa il token al nome del candidato** fino al momento del voto → anonimato preservato

## Disponibilità

È la proprietà che garantisce che **un sistema sia sempre operativo e accessibile** quando serve. Immagina un sito web: se qualcuno riesce a buttarlo giù, magari intasandolo di richieste, quel sito non è più disponibile per gli utenti. Questo è proprio l'obiettivo degli attacchi **DoS (Denial of Service)**.

### Tipi di DoS

- **Saturazione della banda:** il sistema riceve troppe richieste, più di quante ne possa gestire → va in tilt.
- **Saturazione delle risorse:** si consuma la RAM, la CPU, lo spazio disco, ecc.

### Come ci si difende?

- **Accesso solo ad utenti autenticati**, se possibile.
- **Captcha:** evitano richieste automatiche continue.
- **Load balancing:** distribuisce il carico tra più server.
- **Timeout:** chiude connessioni lente o sospette.

I CAPTCHA numerici sono nati per aiutare **Google Street View a leggere** numeri civici difficili da interpretare, **non per censurarli**, ma per **geolocalizzarli meglio**. Ha **censurato** invece **volti e targhe** in Street View.

### Cosa sono i CAPTCHA?

**CAPTCHA** sta per:

Completely Automated Public Turing test to tell Computers and Humans Apart.

È un test **automatico** creato per distinguere **un essere umano da un bot** (cioè un programma automatico).

### Obiettivo principale:

**Impedire che i bot accedano o abusino di un servizio online.**

### Esempi classici di CAPTCHA:

- Scrivere una parola distorta da un'immagine
- Selezionare tutte le immagini con semafori o autobus
- Risolvere semplici operazioni matematiche
- Trascinare un puzzle nel punto corretto

### Cos'è invece reCAPTCHA?

**reCAPTCHA** è una **versione evoluta** dei CAPTCHA, sviluppata inizialmente da un professore della Carnegie Mellon, poi acquistata da **Google**.

### reCAPTCHA ha due scopi:

1. **Bloccare i bot** (come un CAPTCHA normale)
2. **Aiutare i progetti di intelligenza artificiale**, ad esempio:
  - Per **digitalizzare libri** difficili da interpretare da OCR (Optical Character Recognition), come parte di **Google Books**
  - Oggi aiuta a **migliorare i sistemi di riconoscimento immagini** (es. auto, semafori...)

### Perché sono stati inventati?

Sono nati per contrastare **l'abuso automatico di servizi web**, come:

- Registrazioni massive di account fake
- Invio di spam tramite form
- Acquisto automatico di biglietti (ticket scalping)
- Tentativi di attacchi brute-force su login

L'idea è:

**Un umano sa risolverli, un bot no** (o almeno, ci mette molto di più).

---

### E oggi? Con l'IA?

Oggi i CAPTCHA sono **sempre più difficili** perché i bot stanno diventando molto intelligenti (grazie all'AI).

Per questo si usano tecniche più complesse come:

- **Invisible reCAPTCHA** (capisce se sei umano solo osservando come ti muovi nel sito)
- **Behavioral Analysis** (tracciamento del mouse, tempo di digitazione ecc.)

## Controllo di Accesso (o Autorizzazione)

Serve a stabilire **chi può fare cosa**. Non basta sapere *chi sei* (autenticazione), bisogna anche sapere **cosa ti è permesso fare**. Questo si fa attraverso le **policy**, cioè delle regole.

### Esempi di Policy:

1. Puoi leggere file pubblici.
2. Puoi scrivere solo nei file pubblici che possiedi.
3. Non puoi fare il  **downgrade** di un file.
4. Devi cambiare la password quando scade.
5. Se sei un "utente segreto", puoi leggere file non pubblici.
6. Un utente segreto ha il permesso di scrivere su un qualunque file non pubblico
7. Un amministratore ha il permesso di sostituire un qualunque file con una sua versione obsoleta
8. Un utente che non cambia la sua password scaduta (negligente) ha il divieto di compiere qualunque operazione
9. Un utente che non cambia la sua password scaduta (negligente) non ha discrezione di cambiarla

### Ma c'è un problema...

Le policy possono **entrare in conflitto/inconsistenze**. Per esempio:

- **Policy 3** dice che **nessuno può fare il downgrade**.
- **Policy 7** dice che **l'amministratore può farlo**.

Oppure:

- **Policy 8** dice che se non cambi la password, non puoi fare nulla.
- **Policy 9** dice che però **non puoi nemmeno cambiarla** → sei bloccato per sempre!

---

### Come si analizzano le policy?

Per gestirle in modo chiaro si usano **quattro concetti fondamentali**:

- **Ruoli**: utente, utente segreto, amministratore, negligente.
- **Operazioni**: leggere, scrivere, fare downgrade, cambiare password...
- **Modalità**: obbligo, permesso, vietato, discrezionalità.
- **Utente**: ogni entità con un ruolo.

### Modalità spiegate con logica

Si può usare la logica dei predicati per ridurre tutto a un'unica modalità: **Obbligatorio(x)**

Da cui derivano:

- $\text{Vietato}(x) \rightarrow \text{Obbligatorio}(\neg x)$
- $\text{Permesso}(x) \rightarrow \neg \text{Obbligatorio}(\neg x)$
- $\text{Discrezionale}(x) \rightarrow \neg \text{Obbligatorio}(x)$

Questo aiuta a trovare **inconsistenze**:

- $\text{Contraddizione} \equiv \text{Obbligatorio}(x) \wedge \neg\text{Obbligatorio}(x)$
- $\text{Dilemma} \equiv \text{Obbligatorio}(x) \wedge \text{Obbligatorio}(\neg x)$

#### Strategie per evitare inconsistenze

1. **Priorità dei ruoli:** se hai più ruoli, uno può prevalere sugli altri.
  - Es: un ruolo di amministratore batte quello di utente base.
  - Come nei firewall: se non c'è una regola esplicita, si blocca tutto.
2. **Grafi delle policy:**
  - Nodi = utenti, archi = azioni.
  - Se il grafo ha **cicli logici**, ci sono inconsistenze.

#### Esempi di Sistemi di Policy

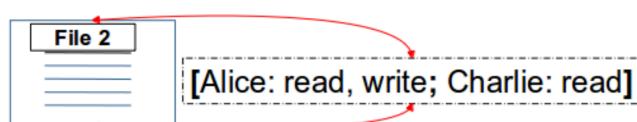
- **MAC (Mandatory Access Control):**
  - Basato su politiche mandatorie e regole fisse e obbligatorie.
  - Nessuno può modificarle, nemmeno gli admin.
  - Usato in ambiti militari o critici e ad alta sicurezza, tipo SELinux.
- **RBAC (Role-Based Access Control):**
  - Basato su politiche non mandatorie.
  - I permessi sono associati a ciascun ruolo.
    - $\text{Permesso}(x) = \neg\text{Vietato}(x)$
    - $\text{Vietato}(x) = \neg\text{Permesso}(x)$
  - Più flessibile, adatto a contesti e sistemi operativi comuni (aziende, siti, app).

#### Metodi per implementare il controllo accessi

1. **ACM (Access Control Matrix):**
  - Matrice con righe = utenti, colonne = file.
  - Una cella  $ACM[s, o]$  indica i permessi di  $s$  su  $o$ , dove  $s$  in Linux è un gruppo di soggetti ed  $o$  è l'oggetto.

	File1	File2	File3	Program1
Alice	<i>read, write</i>	<i>read</i>		<i>execute</i>
Bob	<i>read</i>		<i>read, write</i>	
Charlie		<i>read</i>		<i>read, execute</i>

2. **ACL (Access Control List):**
  - Ogni file ha una lista di chi può farci cosa.
  - Ogni colonna dell'ACM registrata con lo specifico oggetto.



3. **CaL (Capability List):**
  - Invece di essere allegata al file, è **allegata al soggetto**.
  - Ogni utente ha la lista delle risorse su cui ha accesso e cosa può farci.



## Tassonomia di attaccanti

### 1. Chi sono gli attaccanti?

Sono tutte quelle persone o gruppi che vogliono compromettere la sicurezza di un sistema informatico, per vari motivi.

### 2. Come li classifichiamo?

Gli attaccanti si distinguono in base a:

- **Risorse che hanno a disposizione:**

Per esempio, alcuni hanno pochi strumenti (come un hacker alle prime armi), altri invece hanno budget enormi, computer potenti e team dedicati (come servizi segreti o organizzazioni criminali).

- **Esperienza:**

C'è chi ha competenze basse e si affida a tool "preconfezionati", e chi è un esperto di programmazione, crittografia, analisi delle reti.

- **Rischio che sono disposti a correre:**

Alcuni vogliono agire velocemente senza farsi beccare, altri invece sono disposti a rischiare molto perché magari fanno spionaggio industriale o operazioni di intelligence.

### 3. Quali sono i moventi?

Gli attaccanti non attaccano "per noia", ma di solito vogliono ottenere qualcosa, per esempio:

- Ricchezza (rubare soldi o dati da rivendere)
- Informazioni sensibili (dati personali, segreti aziendali)
- Potere o gloria (dimostrare abilità, fare "hacking etico" o malevolo)
- Spionaggio industriale o politico

### 4. Ruoli e tipologie di attaccanti

- **Hacker:**

Possono essere "white hat" (eticamente corretti), "black hat" (criminali) o "grey hat" (mezzo tra i due).

- **Insiders:**

Personne che lavorano dentro l'azienda o organizzazione e che hanno già accesso ai sistemi. Possono essere dipendenti infedeli o persone che involontariamente causano problemi.

- **Spionaggio industriale:**

Attacchi organizzati da aziende concorrenti o stati per rubare segreti industriali.

- **Servizi segreti:**

Organizzazioni governative che fanno intelligence.

- **Organizzazioni criminali:**

Gruppi che fanno estorsioni, frodi e attività illegali.

- **Team difensivi:**

Personne che lavorano per la sicurezza aziendale, testano i sistemi per trovare vulnerabilità.

### 5. Difesa da insiders

Gli **insiders** sono particolarmente pericolosi perché hanno già accesso, quindi si usano principi come:

- **Zero Trust:** non si dà mai fiducia implicita a nessuno, anche se interno

- **Separation of Duty (Separazione dei compiti):** nessuna persona ha tutti i permessi o responsabilità per prevenire abusi

## 6. Perché è importante avere un modello di attaccante?

Quando progettiamo o valutiamo la sicurezza di un sistema o protocollo, dobbiamo capire **contro chi ci stiamo difendendo**.

Un protocollo può essere sicuro contro attaccanti "semplici" ma non contro avversari sofisticati. Quindi si definisce un modello di attaccante con certe capacità e risorse, e si verifica se il protocollo regge sotto quelle condizioni.

### Modello Dolev-Yao

È un modello usato per descrivere il comportamento di un **attaccante ideale** nel contesto di comunicazioni sicure, come ad esempio nei protocolli crittografici.

#### Come funziona?

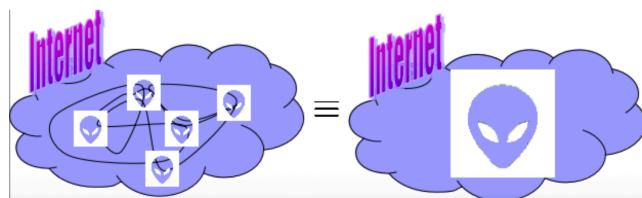
##### 1. Attaccante unico e superpotente

Si immagina un solo attaccante che ha il controllo totale della rete: può **intercettare, modificare, cancellare, e reinviare tutti i messaggi** che viaggiano da un mittente a un destinatario.

In pratica, può fare tutto quello che vuole con il traffico di rete, come se fosse il "padrone" della comunicazione.

##### 2. Perché un attaccante unico?

Anche se in realtà potresti avere più attaccanti sparsi nella rete, si dimostra che un insieme di attaccanti collusi equivale a un solo attaccante potentissimo che ha lo stesso controllo su tutto.



##### 3. Limite importante: non può rompere la crittografia

Il modello assume che la crittografia usata sia **perfetta** o almeno **non violabile** dall'attaccante. Quindi, se un messaggio è cifrato correttamente, l'attaccante non può leggerne il contenuto, né decifrarlo.

Questo limite è fondamentale per poter valutare i protocolli in modo chiaro: se il protocollo "cade", non è colpa della crittografia (che si presume sicura), ma di come è progettato il protocollo stesso.

#### Perché è importante?

- Il Modello Dolev-Yao è una base teorica per analizzare la sicurezza dei protocolli di comunicazione.
- Ti permette di ragionare su tutte le possibili manipolazioni e interferenze che un attaccante superpotente può fare, senza dover pensare a rompere la crittografia.
- Così si possono trovare vulnerabilità nei protocolli, come errori di progettazione, attacchi replay, manomissione di messaggi, ecc.

### Modello General Attacker

Il modello GA è nato per descrivere un tipo di attaccante più "realistico" rispetto al modello Dolev-Yao, pensando al mondo tecnologico moderno.

#### Come funziona?

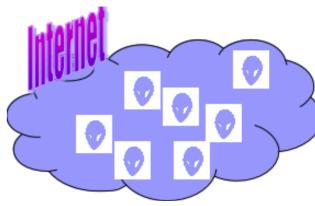
##### 1. Attaccanti singoli e non collusi

Qui si assume che ogni attaccante agisca da solo, senza coordinarsi o colludere con altri.

Quindi non c'è un "super-attaccante" che controlla tutta la rete, ma tanti piccoli attaccanti sparsi, ognuno con i propri obiettivi e limiti.

##### 2. Nessuna collusione

Ogni attaccante agisce per conto suo, senza cooperare con altri. Questo riflette più fedelmente il panorama attuale, dove potresti trovarsi di fronte a tanti hacker, insider o utenti malintenzionati con motivazioni diverse.



### 3. Non rompe la crittografia

Come nel modello Dolev-Yao, anche il General Attacker non è capace di rompere la crittografia usata.

Quindi, se un messaggio è cifrato bene, non potrà leggerlo o modificarlo con successo.

### 4. Attacchi meno "potenti" ma più realistici

Poiché non c'è un attaccante unico superpotente che domina tutta la rete, gli attacchi sono più localizzati e meno devastanti rispetto a quelli immaginati dal modello Dolev-Yao.

---

#### Perché è stato creato?

- Nel 2003, con l'evoluzione delle reti e la diffusione massiccia di utenti e dispositivi, si è capito che l'idea di un attaccante "onnipotente" che controlla tutto è troppo irrealistica.
- Oggi, la sicurezza deve difendersi da tante minacce diverse, spesso indipendenti e poco coordinate.
- Quindi GA vuole riflettere meglio la varietà degli attaccanti reali, con diversi obiettivi e capacità.

## Autenticazione

L'autenticazione è quel processo che serve per **verificare l'identità di un'entità** (persona, macchina, servizio). In pratica: "Sei davvero chi dici di essere?"

Ci sono **quattro scenari tipici** in cui l'autenticazione può avvenire:

1. **Utente-computer**: è il più comune. Tu che accedi con una password o impronta al tuo PC o telefono.
2. **Computer-computer**: per esempio, due server che devono parlarsi in sicurezza (si scambiano certificati, IP, MAC address).
3. **Computer-utente**: tipo tu che accedi a un sito e devi assicurarti che **sia proprio quel sito** e non uno fasullo (pensiamo al lucchetto HTTPS nel browser).
4. **Utente-utente**: molto raro. Un esempio è **Kerberos**, dove due utenti si autenticano reciprocamente tramite un intermediario fidato.

---

#### Tipi di autenticazione

Il principio base è questo: per autenticarti puoi usare **una o più delle seguenti categorie**:

- **Qualcosa che conosci** (una password, un PIN)
- **Qualcosa che possiedi** (una chiavetta, una carta, un token)
- **Qualcosa che sei** (impronta, volto, voce)

Vediamoli uno a uno con i rischi e contromisure:

## Autenticazione basata sulla conoscenza

Esempio classico: inserisci una password.

#### Rischi:

- **Guessing**: qualcuno indovina la tua password (magari è "123456"...).
- **Shoulder surfing (snopping)**: qualcuno ti guarda mentre digitri la password.
- **Spoofing**: ti fanno inserire la password in un sito falso.
- **Sniffing**: la password viene intercettata nel traffico di rete.

#### Tipici attacchi:

- **Attacco standard:** provano con password comuni ("admin", "password").
- **Dizionario:** provano tutte le parole di un dizionario.
- **Forza bruta:** provano tutte le combinazioni possibili di caratteri.

#### **Contromisure:**

- Password robuste (non banali, lunga, con numeri e simboli).
- Limitare il numero di tentativi (blocca temporanei).
- CAPTCHA (per evitare che i bot automatizzino i tentativi).
- Non salvare password in chiaro! Si usano gli **hash**.

**Curiosità:** il **NIST** nel 2004 impose password complesse. Ma nel 2016 cambiò approccio, perché la troppa complessità spingeva gli utenti a scriverle in giro (es. su post-it o file .txt). Quindi oggi si prediligono **password lunghe, semplici ma uniche**, magari con altri sistemi di sicurezza a supporto.

## **Autenticazione basata sul possesso**

| Esempio: carta di credito, chiave USB, smart token, OTP.

#### **Rischi:**

- Può essere **perso**.
- Può essere **ceduto volontariamente**.

#### **Soluzione? Autenticazione a due fattori (2FA):**

| Combina qualcosa che possiedi con qualcosa che conosci (es. token + PIN).

#### **OTP (One Time Password):**

È una password **valida solo una volta** e per un **tempo limitato**. Anche se qualcuno te la ruba, **non può riutilizzarla**.

Due tipi di token:

1. **Con tastiera:** devi inserire una password prima di ricevere l'OTP.
2. **Senza tastiera:** ti mostra direttamente l'OTP.

Sembrano diversi, ma sono **equivalenti** in sicurezza: cambia solo dove viene inserita la password (prima nel token o dopo nel sito).

Questi token sono **sincronizzati col server**, ma non richiedono una rete: usano un algoritmo che genera OTP basandosi sul tempo e su una chiave segreta condivisa.

## **Autenticazione biometrica**

| Usa caratteristiche uniche del tuo corpo (fisiche o comportamentali).

#### **Tipi:**

- **Fisiologiche:** impronte, iride, volto, voce.
- **Comportamentali:** firma, passo, modo di digitare.

#### **Funziona in due fasi:**

1. **Enrollment:** il sistema ti "studia" e salva le tue caratteristiche biometriche sotto forma di template.
2. **Authentication:** quando ti autentichi, confronta ciò che legge al momento con quanto salvato.

| Se la somiglianza supera una soglia → accesso concesso!

#### **Esempio impronte digitali:**

- Pattern unici come **loop, arch, whorl**.

- Si analizzano le **minuzie**: punti dove le linee si biforciano o finiscono.

#### Vantaggi:

- Facile da usare, veloce.
- Non serve ricordare niente.

#### Limiti:

- **Non infallibile**: possibile riconoscere un altro come te (falsi positivi) o non riconoscerti (falsi negativi).
- Se ti rubano un'impronta, **non puoi cambiarla come fai con una password!**

Ogni metodo di autenticazione ha pro e contro. I sistemi più sicuri sono quelli che combinano **più fattori** (conoscenza + possesso, o possesso + biometria), come nei portali bancari moderni.

Ecco un esempio reale:

Bancomat:

- *Conoscenza*: il PIN
- *Possesso*: la carta fisica

Home banking:

- *Conoscenza*: password
- *Possesso*: token o codice via SMS/APP

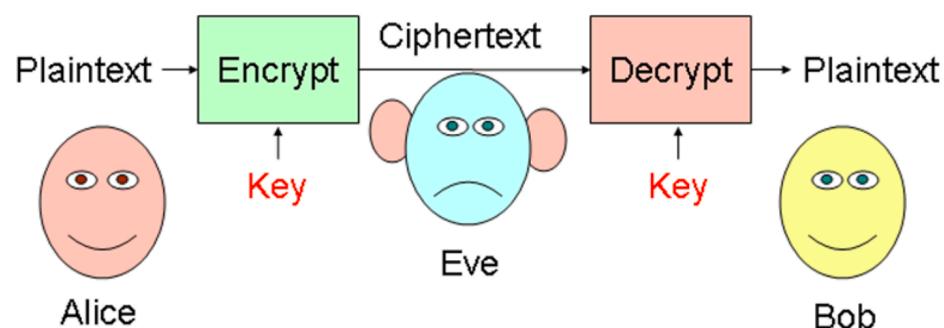
## Cenni di crittografia

La **crittografia** è la disciplina che si occupa di **proteggere i messaggi**, trasformandoli in un formato illeggibile (critttesto) che solo chi possiede la **chiave giusta** può riportare al significato originale (testo in chiaro).

Quindi:

- **Crittare / cifrare** = codificare il messaggio
- **Decrittare / decifrare** = riportarlo in chiaro

L'obiettivo è semplice: **modificare la forma del messaggio (sintassi)** ma **non il suo significato (semantica)**. Se decifro con la **chiave corretta**, recupero il messaggio originale. Se uso una chiave a caso (diversa da quella corretta), ottengo solo rumore.



### Tipi di crittografia

#### 1. Crittografia simmetrica

| La stessa chiave viene usata per cifrare e decifrare.

Immagina un lucchetto con una singola chiave: la stessa che lo chiude, lo apre. Esempi famosi: **AES, DES**.

#### Pro:

- Molto veloce ed efficiente.
- Ottima per grandi quantità di dati.

### Contro:

- Il problema più grande è: **come ci scambiamo la chiave segreta?**
- Se la chiave finisce in mani sbagliate, è finita: chiunque può leggere tutto.

### Concetto importante:

Ogni coppia di utenti (es. A e B) ha bisogno di una chiave **dedicata**, detta  $K_{ab}$ , valida solo per la loro comunicazione.

Però:  $K_{ab}$  **deve essere scambiata in modo sicuro!** Ed è proprio qui che entra in gioco...

## 2. Crittografia asimmetrica

Si usano due chiavi distinte: una pubblica e una privata.

Funziona così:

- Ogni utente genera una **coppia di chiavi**:
  - **Chiave pubblica ( $K_a$ )** → la puoi dare a chiunque.
  - **Chiave privata ( $K_a^{-1}$ )** → la tieni segreta.

Le due chiavi sono matematicamente **inverse**:

- Se cifro con  $K_a$  → decifro solo con  $K_a^{-1}$
- Se cifro con  $K_a^{-1}$  → decifro solo con  $K_a$

### Esempio:

- Se Alice vuole mandare un messaggio a Bob:
  - Prende la **chiave pubblica di Bob ( $K_b$ )** e cifra il messaggio.
  - Solo Bob, con la **sua chiave privata ( $K_b^{-1}$ )**, potrà leggerlo.

**Abbiamo confidenzialità**: solo Bob può leggere.

### Scenari pratici con crittografia asimmetrica

Immagina Alice e Bob, ognuno con la sua coppia di chiavi. Cosa può fare Alice?

#### 1. Alice cifra con la sua chiave privata ( $K_a^{-1}$ )

- Chiunque può decifrare con la **sua chiave pubblica ( $K_a$ )**.
- Quindi:
  - **SI Autenticazione** (solo Alice ha  $K_a^{-1}$  → il messaggio viene davvero da lei)
  - **NO confidenzialità** (tutti hanno  $K_a$  → tutti possono leggere)

#### 2. Alice cifra con la chiave pubblica di Bob ( $K_b$ )

- Solo Bob, con  $K_b^{-1}$ , può decifrare.
- Quindi:
  - **SI Confidenzialità**
  - **NO autenticazione** (chiunque può usare la chiave pubblica di Bob per inviare)

#### 3. Alice cifra prima con la sua privata, poi con la pubblica di Bob

- Bob, per leggerlo, dovrà prima usare la **sua privata ( $K_b^{-1}$ )**, poi la **pubblica di Alice ( $K_a$ )**.
- Quindi abbiamo:
  - **SI Confidenzialità** (solo Bob può leggerlo)
  - **SI Autenticazione** (solo Alice poteva firmarlo con la sua privata)

Questo è il **doppio strato perfetto**: confidenzialità + autenticazione.

### Problema critico: l'associazione chiave ↔ proprietario

Tutto bello finché...

Come faccio a sapere che una chiave pubblica appartiene davvero a un certo utente?

Se un attaccante si spaccia per Bob e mi manda **la sua chiave pubblica finta**, io cifro con quella pensando che sia quella di Bob. Risultato? Il messaggio è protetto, ma **verso l'attaccante!**

#### Soluzione? I certificati digitali

- Sono dei documenti firmati da un'autorità fidata (CA, Certification Authority).
- Confermano che **quella chiave pubblica appartiene veramente** a un certo soggetto.

Senza questa verifica, l'intero meccanismo di autenticazione cade a pezzi.

## Hash crittografico

Una **funzione di hash** prende in input **un messaggio di lunghezza arbitraria** e lo trasforma in una **stringa (digest)** di **lunghezza fissa**.

Esempio:

```
Input: "Ciao Emanuel!"  
Output: "d52ab8f92f4a28..." (lunghezza fissa, tipo 256 bit)
```

Una buona funzione di hash è **one-way**: ti dà un'impronta digitale del messaggio, ma **non puoi risalire al messaggio originale** da quell'impronta.

#### Le 4 proprietà fondamentali di una funzione hash "sicura"

##### 1. Facilità di calcolo

Dato un messaggio, è **veloce** calcolare il suo hash.

##### 2. Pre-immagine resistente (one-way)

Se conosci solo l'hash, **non puoi risalire** al messaggio originale.

(es. conosci  $H = sha256("password123")$  ma non puoi ottenere "*password123*" da  $H$ )

##### 3. Seconda pre-immagine resistente

Sia computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale a quello di una data stringa.

##### 4. Collisione resistente

Sia computazionalmente intrattabile la ricerca di una coppia di stringhe in input che diano lo stesso hash.

Queste proprietà non sono "magia", ma sono **obiettivi di progetto**. La vera sfida è **costruire algoritmi di hash** che le rispettino il più possibile.

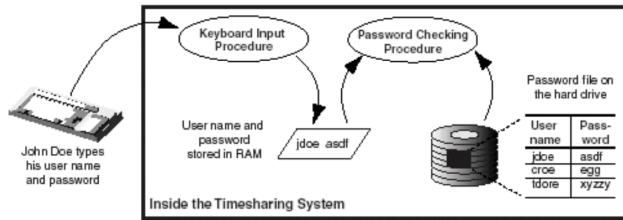
#### Perché l'hashing è importante per la sicurezza?

L'hash è usato in tantissimi ambiti:

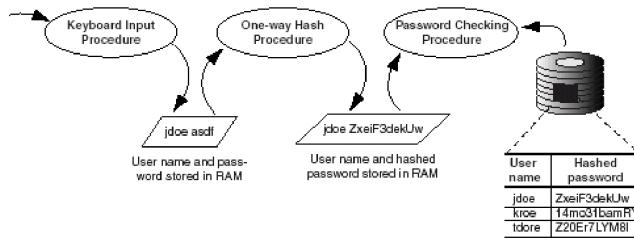
- Autenticazione delle password
- Verifica dell'integrità dei file (es. hash SHA256 su un download)
- Firma digitale (ci si firma sull'hash del documento, non sul documento intero)

#### Esempio storico: CTSS

Negli anni '60, nel sistema operativo **CTSS** (Compatible Time Sharing System, MIT), non si usava ancora l'hashing per proteggere le password: si salvava la password in chiaro su un file di sistema protetto da politica di sicurezza. Ci furono numerosi attacchi registrati.



Nel '67 viene potenziato alla Cambridge University con una funzione hash. Il file delle password memorizza l'hash di ciascuna password.



### Ma... l'hash non è invincibile!

Anche se **non è invertibile**, ci sono modi per **indovinare** il messaggio che ha generato un certo hash.

#### Attacchi comuni:

- **Brute-force**: provo tutte le password finché non trovo quella giusta.
- **Rainbow table**: pre-calcolo miliardi di password comuni e i loro hash, così da poterli confrontare velocemente.

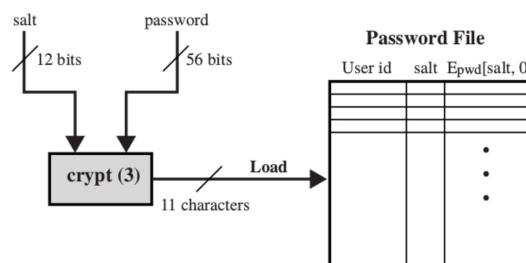
### Il salting: la difesa intelligente

Per contrastare questi attacchi, si usa il **salting**:

Il salt è un valore casuale aggiunto alla password prima di calcolare l'hash.

#### Come funziona (es. in Unix):

1. Quando l'utente crea la password, il sistema **genera un salt randomico**
2. Si prende la password + salt, e si usa per **criptare una stringa di zeri**
3. Si usa una funzione come `crypt(3)` (basata su DES) per calcolare l'hash



- Nessun utilizzo di hashing
- Chiave di 56 bit ricavata dai 7 bit meno significativi dei primi 8 caratteri della password
- Viene criptato salt e stringa di zeri
- Originale uso del segreto da proteggere come chiave invece che come testo in chiaro. Per forza!

#### Obiettivo:

- Due utenti con la **stessa password** avranno hash **diversi**, perché i salt sono diversi.
- Rende inutili le rainbow table pre-generate.

## Differenza tra hashing e crittografia

Anche se possono sembrare simili (entrambi trasformano il messaggio), c'è una **differenza fondamentale**:

	Hashing	Crittografia
<b>Obiettivo</b>	Verifica, impronta	Protezione e segretezza
<b>Reversibilità</b>	Non si può tornare indietro	Si può decriptare con la chiave
<b>Uso tipico</b>	Password, integrità, firme	Messaggi segreti, canali sicuri
<b>Confronto</b>	Si confronta l'hash (es. login)	Si decifra con la chiave

L'hash **non si decripta**. Si confronta.

Esempio: quando fai login, il sistema non conosce la tua password. Prende quella che inserisci, ne calcola l'hash, e lo confronta con quello salvato.

### Hash da funzione crittografica?

`crypt(3)` usa un cifrario (DES) per creare un "hash". In effetti:

Una funzione di cifratura può essere usata per creare una funzione di hash, se ben progettata.

Ma attenzione: **non sono la stessa cosa**. Le funzioni hash sono pensate per essere **non invertibili**, mentre quelle crittografiche **devono essere invertibili** (con la chiave giusta).

Quindi riassumendo:

- `crypt(3)` è una **funzione di hashing** pensata per salvare password in modo sicuro.
- Anche se **usa DES**, lo fa in modo **irreversibile**, come se fosse una funzione di hash.
- **Linux salvava l'hash + salt** nel file `/etc/passwd` (leggibile da tutti), e **non la password**.
  - Nota: Oggi Linux salva hash + salt degli utenti nel file `/etc/shadow`, più sicuro e **accessibile solo da root**.
- Durante il login, **riapplica l'hashing con lo stesso salt** per verificare se coincide.

### Perché `crypt(3)` usa DES?

Negli anni '70 **non esistevano ancora funzioni di hash sicure per le password**, quindi si è riciclato DES (che era uno standard di cifratura) usandolo in modo "creativo" come funzione di hashing.

### Perché non si usava una vera funzione di hash?

Perché:

- SHA-1, SHA-2, ecc. **non esistevano**.
- MD5/MD4 erano ancora in fase iniziale e troppo **veloci** (quindi insicure).
- Le funzioni hash non erano pensate per difendere le password da attacchi brute-force.

### Perché è stato usato a lungo?

Per **compatibilità storica**: `crypt(3)` era già integrato nei sistemi Unix/Linux, e cambiarlo avrebbe richiesto riscrivere tanti software.

### Perché oggi non si usa più DES?

Perché:

- DES è **debole** (chiave corta, facilmente attaccabile).
- Oggi abbiamo algoritmi **molto più sicuri** per le password: `bcrypt`, `scrypt`, `Argon2`.

## Freshness

In una parola, **freshness = novità**.

Significa che un messaggio, una chiave o un token è **recente, appena generato, e non riutilizzato**.

Perché ci interessa? Perché se un messaggio è fresco, allora:

- non è stato **intercettato in passato** e riproposto da un attaccante (attacco *replay*);
  - è legato ad un momento preciso → quindi puoi fidarti del fatto che sia "**valido ora**" e **non in un altro momento**.
- 

#### A cosa serve la freshness?

La freshness aiuta a garantire **due cose fondamentali**:

1. **Autenticazione** → perché ogni autenticazione è unica e nuova, quindi **non riutilizzabile**.
  2. **Segretezza (a lungo termine)** → perché se un messaggio non è "fresco", può essere riusato o analizzato da un attaccante nel tempo. Anche un algoritmo sicuro oggi può diventare vulnerabile in futuro, quindi è bene non riutilizzare nulla.
- 

#### E se non c'è freshness? → Attacchi Replay

Un attaccante può **intercettare un messaggio valido** e **rimandarlo al destinatario in un altro momento**, anche se il mittente non l'ha più inviato.

Esempio pratico:

- Tu invii a un server: "Ehi, autorizzami! Ecco le credenziali!"
- L'attaccante salva il messaggio.
- Domani lo rimanda al server → e il server ci casca, ti fa accedere di nuovo!

Se quel messaggio fosse **fresco**, il server si accorgerebbe che è **vecchio o già visto** e lo scarterebbe.

## Protocolli basilari per la freshness

#### Come si garantisce la freshness?

Due meccanismi molto comuni:

---

#### Timestamp

Un **timestamp** è un marcitore temporale e **protegge chi vuole dare garanzia di freshness**, tipo:

2025-07-04 15:26:45

#### Vantaggi:

- Facile da usare e verificare.
- Puoi dire "questo messaggio è valido per 5 minuti".

#### Svantaggi:

- Devi **fidarti dell'orologio** dell'altro (es. del server) e controllare che siano sincronizzati.
  - Se il server è malintenzionato o ha un orologio sbagliato, ti può fregare.
  - Non protegge bene l'utente, ma più chi offre il servizio.
- 

#### Nonce

Un **nonce** è un numero **randomico** e **unico** usato **una volta sola**. **Protegge chi vuole ricevere garanzia di freshness**.

#### Vantaggi:

- È creato **dall'utente stesso** → quindi serve a **proteggere chi invia il messaggio** (l'utente).
- Non serve sincronizzazione di orologio.

#### Attenzione:

- Deve essere **veramente casuale** e **mai riutilizzato**.
- Se è prevedibile o già usato, può essere sfruttato da un attaccante.

Esempio di uso:

1. Alice vuole inviare un messaggio a Bob.
2. Genera un nonce casuale N123456 .
3. Invia il messaggio insieme al nonce → magari cifrati insieme.

4. Bob riceve il tutto e salva **N123456**.
5. Se qualcuno riprova a inviare lo stesso messaggio con lo stesso nonce → Bob se ne accorge e rifiuta.

#### Cosa succede se combini freshness + crittografia?

Ottieni un messaggio **unico, irripetibile e sicuro**.

Perché se qualcuno prova a inviare lo stesso messaggio:

- Il nonce è già stato usato → il messaggio è scartato.
- Il contenuto è cifrato con il nonce → quindi l'attaccante **non può nemmeno capirlo o modificarlo**.

In pratica:

Messaggio + Nonce → Cifrati insieme → Sicurezza + Freshness

#### Protocolli basilari di autenticazione senza freshness

##### 1. Simmetrico

- **Prerequisito:** A e B condividono una chiave segreta  $K_{AB}$ .
- **Protocollo:**

$$A \rightarrow B : M, MAC(M, K_{AB})$$

Qui A invia un messaggio M autenticato tramite  $MAC$  con la chiave condivisa. B può verificare chi ha inviato il messaggio (autenticazione), ma **non** può distinguere se il messaggio è "fresco" o una ripresa (vulnerabile a replay).

##### 2. Asimmetrico (firma digitale)

- **Prerequisiti:** A possiede chiave privata  $K_A^{-1}$ , B dispone della corrispondente chiave pubblica  $K_A$ .
- **Protocollo:**

$$A \rightarrow B : M, Sign_A(M)$$

A invia M e la sua firma digitale. B verifica la firma con  $K_A$  e ottiene autenticazione e integrità. Anche qui manca freshness (attacco replay possibile).

#### Protocolli basilari di autenticazione con freshness

##### 1. Simmetrico con challenge/nonce

- **Prerequisiti:** chiave condivisa  $K_{AB}$ .
- **Protocollo:**

1. B genera una nonce  $N_B$  e la invia in chiaro:

$$B \rightarrow A : N_B$$

2. A risponde autenticando la nonce:

$$A \rightarrow B : MAC(N_B, K_{AB})$$

- **Vantaggi:** B si assicura che la risposta è stata generata da A al momento, dato che solo A possiede  $K_{AB}$  e la nonce non è stata usata prima. Garantisce **autenticazione + freshness**, protegge da attacchi replay.

##### 2. Asimmetrico con challenge/nonce

- **Prerequisiti:** A e B hanno i propri certificati e coppie di chiavi.
- **Protocollo:**

1.  $B \rightarrow A : N_B$

2.  $A \rightarrow B : \{N_B, A\}_{K_A^{-1}}$

(A firma la nonce insieme alla propria identità)

- **Vantaggi:** B verifica la firma con  $K_A$  e ottiene **autenticazione, integrità e freshness** (la nonce era nuova). B può confermare l'identità di A e che la risposta è recente.

## Sintassi dei messaggi crittografici

### Atomici

- Nomi di agenti:  $A, B, C, \dots$
- Chiavi crittografiche:  $k_a, k_b, \dots, k_a^{-1}, k_b^{-1}, \dots$   
 $k_{ab}, k_{ac}, \dots$
- Nonce:  $N_a, N_b, \dots$
- Timestamp:  $T_a, T_b, \dots$
- Digest:  $h(m), h(n), \dots$
- Etichette: "Trasferisci £100 dal conto di..."

### Composti

- Concatenazioni:  $m, m'$ 
  - ciascuno può essere crittotesto
- Crittotesti:  $m_k$ 
  - il testo in chiaro può essere concatenazione

### Problema

Si può autenticare un messaggio concatenato?

Quando si parla di *sintassi* qui, non si intende la grammatica dei linguaggi di programmazione, ma **la struttura e la composizione dei messaggi cifrati nei protocolli di sicurezza**.

In particolare:

- Come i messaggi sono costruiti (concatenati).
- Come vengono interpretati durante l'autenticazione.
- Cosa può andare storto se non li strutturi bene.

### Il problema della concatenazione semplice

Supponiamo tu abbia due messaggi:

- $M_1 = \text{"Sono Alice"}$
- $M_2 = \text{"Voglio autenticarmi"}$

Se li concateni così:  $M_1 || M_2$  e poi li cifri insieme:

$$C = Enc_K(M_1 || M_2)$$

tu pensi: *bene, ho autenticato tutto!*

Ma in realtà **non è sufficiente**, perché:

La crittografia di una concatenazione non garantisce automaticamente che i singoli pezzi siano autentici e ben separati.

#### Perché?

Perché **un attaccante potrebbe prendere  $M_1$  da Alice e  $M_2$  da un altro utente**, concatenarli, cifrarli con una chiave qualsiasi (magari anche la stessa), e **il destinatario non potrebbe accorgersi che il messaggio è una composizione fraudolenta**.

#### La chiave del problema

La **cifratura non è univoca rispetto alla struttura interna del messaggio**, quindi:

- Se il messaggio è composto male (tipo  $M_1 || M_2$ ), un attaccante potrebbe **manipolarlo**.
- Non è detto che chi riceve riesca a **verificare chi ha scritto cosa**, anche se riesce a decifrare tutto.

#### L'autenticazione non è legata ai nomi

"I nomi degli agenti servono solo a noi per capire chi manda cosa..."

Questo significa:

- Quando analizziamo un protocollo, scriviamo spesso qualcosa tipo:

$A \rightarrow B : \{M\}_K$

e ci sembra ovvio che  $A$  ha mandato quel messaggio a  $B$  cifrato con la chiave  $K$ .

- Ma per un attaccante (e per il sistema automatico), non è detto che ci sia un legame evidente tra il nome dell'agente e il contenuto del messaggio.

Quindi?

Se non includi nel messaggio un'indicazione esplicita del mittente o del contesto, il destinatario non può sapere con certezza da chi proviene il messaggio, anche se la cifratura è corretta.

#### Soluzione: strutturare bene i messaggi

Quando si progettano messaggi sicuri, bisogna includere in modo chiaro e non ambiguo:

- Chi lo manda ( $A$ )
- Chi lo riceve ( $B$ )
- Cosa contiene (messaggio, nonce, timestamp...)
- Come è cifrato e con quale chiave

#### Esempio corretto:

$A \rightarrow B : Enc_K("A", "B", \text{Nonce}, \text{Timestamp})$

Così:

- B sa che il messaggio è destinato a lui.
- Sa che viene da A (perché è scritto dentro).
- Sa che è fresco (nonce/timestamp).
- Può fidarsi della sua autenticità (perché cifrato con chiave giusta).

#### Ha senso usare sia nonce che timestamp?

Sì, ha senso usare timestamp e nonce insieme, soprattutto per rafforzare la sicurezza e garantire freshness da entrambe le parti. Ma non è sempre necessario: dipende dal livello di fiducia nei clock e nella gestione delle challenge.

## Protocolli basilari per la segretezza

### Obiettivo

Garantire che solo A (mittente) e B (destinatario) possano leggere il messaggio  $m$  scambiato tra loro, mantenendo quindi la **segretezza**.

#### Segretezza del messaggio $m$ per $A$ e $B$

- Crittografia simmetrica
  - Prerequisito: chiave  $k_{ab}$  sia condivisa fra  $A$  e  $B$  soli
    1.  $A \rightarrow B : m_{k_{ab}}$

- Crittografia asimmetrica
  - Prerequisito1:  $B$  abbia una chiave privata valida (*sicura, non scaduta*)
  - Prerequisito2:  $A$  possa verificare che  $k_b$  è di  $B$ 
    1.  $A \rightarrow B : m_{k_b}$

#### Si noti che serve certificazione

### 1. Crittografia Simmetrica

### Cos'è?

- Entrambi, A e B, usano la **stessa chiave segreta condivisa**, chiamata  $k_{ab}$ .
- Solo loro due conoscono questa chiave.

### Come funziona?

- A cifra il messaggio **m** usando la chiave segreta  $k_{ab}$ .

- Poi manda a B il messaggio cifrato:

$$A \rightarrow B : m_{k_{ab}}$$

dove  $m_{k_{ab}}$  significa "messaggio m cifrato con la chiave  $k_{ab}$ ".

### Prerequisito fondamentale

- La chiave  $k_{ab}$  deve essere **condivisa solo tra A e B**. Se qualcun altro la conosce, può leggere il messaggio.
- Quindi la segretezza dipende da quanto è sicura la condivisione e la protezione di questa chiave.

## 2. Crittografia Asimmetrica

### Cos'è?

- Qui ogni partecipante ha **due chiavi diverse**:
  - Una chiave pubblica ( $k_b$ ) che può essere conosciuta da tutti.
  - Una chiave privata ( $k_b^{-1}$ ) che deve rimanere segreta e conosciuta solo dal proprietario, in questo caso B.

### Come funziona?

- A prende il messaggio **m** e lo cifra usando la **chiave pubblica di B** ( $k_b$ ).
- Quindi invia a B il messaggio cifrato:

$$A \rightarrow B : m_{k_b}$$

dove  $m_{k_b}$  è il messaggio cifrato con la chiave pubblica di B.

- Solo B può decifrarlo usando la sua chiave privata ( $k_b^{-1}$ ).

### Prerequisiti fondamentali

#### 1. La chiave privata di B deve essere valida e sicura:

Deve essere segreta, non compromessa o scaduta.

#### 2. A deve essere sicuro che $k_b$ è davvero la chiave pubblica di B:

Perché altrimenti potrebbe cifrare il messaggio per qualcun altro (man-in-the-middle).

Per questo serve una **certificazione** (ad esempio tramite un certificato digitale) che attesti che  $k_b$  appartiene veramente a B.

### Perché sono importanti i prerequisiti?

Se la chiave simmetrica non è segreta, o la chiave privata di B è compromessa, o se A non è sicuro della vera identità di B (cioè non verifica la chiave pubblica), allora il protocollo non garantisce la segretezza.

## Protocolli basilari per l'autenticazione

### Obiettivo dell'autenticazione

Far sì che **B sia sicuro che il messaggio o la comunicazione provenga veramente da A**, cioè che A sia davvero chi dice di essere.

## Autenticazione di A con B

### ■ Crittografia simmetrica

- Prerequisito1: chiave  $k_{ab}$  sia condivisa fra A e B soli
- Prerequisito2: B possa verificare il Prerequisito1

1.  $A \rightarrow B : "Sono\ io!"_{k_{ab}}$

### ■ Crittografia asimmetrica

- Prerequisito1: A abbia una chiave privata valida
- Prerequisito2: B possa verificare che  $k_a$  è di A

1.  $A \rightarrow B : "Sono\ io!"_{k_a^{-1}}$

Si potrebbe usare un qualunque testo intelligibile

### 1. Autenticazione con crittografia simmetrica

#### Come funziona?

- A e B condividono una chiave segreta  $k_{ab}$  (prerequisito 1).
- Solo loro conoscono questa chiave.
- A invia a B un messaggio che dimostra di conoscere la chiave segreta (ad esempio cifrando o firmando un messaggio standard come "Sono io!").

#### Esempio di messaggio:

- A manda a B:  
 $"Sono\ io!"_{k_{ab}}$   
cioè il messaggio "Sono io!" cifrato con la chiave condivisa  $k_{ab}$ .

#### Perché funziona?

- Solo A e B conoscono la chiave  $k_{ab}$ .
- Se B riesce a decifrare correttamente il messaggio, sa che chi lo ha inviato conosce la chiave, quindi deve essere A.

#### Prerequisiti importanti:

- **$k_{ab}$  deve essere segreta** e condivisa solo tra A e B.
- **B deve essere certo che  $k_{ab}$  è condivisa solo con A** (prerequisito 2).

### 2. Autenticazione con crittografia asimmetrica

#### Come funziona?

- A ha una coppia di chiavi:
  - Chiave privata ( $k_a^{-1}$ ), che solo A conosce.
  - Chiave pubblica ( $k_a$ ), nota a tutti, compreso B.
- A "firma" un messaggio (anche un semplice testo come "Sono io!") con la sua chiave privata.
- A invia a B:  
 $"Sono\ io!"_{k_a^{-1}}$   
cioè il messaggio firmato con la chiave privata di A.

#### Perché funziona?

- Solo A può creare questa firma digitale perché solo lui ha la chiave privata.
- B verifica la firma usando la chiave pubblica di A ( $k_a$ ).
- Se la verifica funziona, B è certo che il messaggio è stato mandato da A (o almeno da chi ha la chiave privata di A).

### Prerequisiti importanti:

- La chiave privata di A deve essere valida e segreta.
- B deve essere sicuro che la chiave pubblica  $k_a$  appartiene veramente ad A (prerequisito 2).

### Nota importante

Il messaggio "Sono io!" è un esempio semplice. In realtà, il contenuto in sé non ha importanza, perché chiunque potrebbe scrivere "Sono io!". L'autenticazione non avviene tramite il messaggio in chiaro, ma tramite la **prova che solo A può generare** (cioè conoscere la chiave segreta o firmare con la chiave privata).

## Combinare segretezza e autenticazione

### Obiettivo

Vogliamo **inviare un messaggio segreto da A a B, dimostrando che è proprio A a inviarlo** (quindi **segretezza + autenticazione**).

Segretezza del messaggio  $m$  per  $A$  e  $B$ , autenticazione di  $A$  con  $B$

- Crittografia simmetrica
  - Prerequisito1: chiave  $k_{ab}$  sia condivisa fra  $A$  e  $B$  soli
  - Prerequisito2:  $B$  possa verificare il Prerequisito1
- 1.  $A \rightarrow B : m_{k_{ab}}$
- Crittografia asimmetrica
  - Prerequisito1:  $B$  abbia una chiave privata valida
  - Prerequisito2:  $A$  possa verificare che  $k_b$  è di  $B$
  - Prerequisito3:  $A$  abbia una chiave privata valida
  - Prerequisito4:  $B$  possa verificare che  $k_a$  è di  $A$
- 1.  $A \rightarrow B : \{m_{k_a^{-1}}\}_{k_b}$  o 1.  $A \rightarrow B : \{m_{k_b}\}_{k_a^{-1}}$

## CRITTOGRAFIA SIMMETRICA

### Prerequisiti

1. A e B condividono **una chiave segreta**  $k_{ab}$ , conosciuta solo da loro.
2. B deve essere sicuro che solo A e B conoscano  $k_{ab}$ .

### Protocollo

- $A \rightarrow B : m_{k_{ab}}$   
A cifra il messaggio  $m$  con la chiave simmetrica  $k_{ab}$ .

### Cosa otteniamo?

- **Segretezza**: solo B può leggere il messaggio perché solo lui ha  $k_{ab}$ .
- **Autenticazione**: se B riceve qualcosa cifrato con  $k_{ab}$ , può supporre che lo abbia mandato A, perché solo A e B hanno la chiave.

Nota: **l'autenticazione e la segretezza qui non sono fortissime**: se  $k_{ab}$  viene compromessa, chiunque può inviare o leggere messaggi fingendosi A o B.

## CRITTOGRAFIA ASIMMETRICA

### Prerequisiti

1. **B ha una chiave privata valida ( $k_b^{-1}$ )**.
2. **A sa che la chiave pubblica di B ( $k_b$ ) è veramente di B**.
3. **A ha una chiave privata valida ( $k_a^{-1}$ )**.
4. **B sa che la chiave pubblica di A ( $k_a$ ) è veramente di A**.

---

**Due modi per combinare segretezza e autenticazione:**

**Modo 1:**

$$A \rightarrow B : \{m_{k_a^{-1}}\}_{k_b}$$

1. A firma il messaggio con **la sua chiave privata**  $k_a^{-1}$  → questo garantisce **autenticazione** (solo A poteva firmarlo).
2. Poi cifra tutto con **la chiave pubblica di B**  $k_b$  → questo garantisce **segretezza** (solo B può decifrarlo).

**Modo 2:**

$$A \rightarrow B : \{m_{k_b}\}_{k_a^{-1}}$$

1. A cifra  $m$  con **la chiave pubblica di B**  $k_b$  → **segretezza**.
2. Poi firma il tutto con **la sua chiave privata**  $k_a^{-1}$  → **autenticazione**.

**In pratica non cambia molto l'ordine**, perché una delle due operazioni serve a garantire segretezza (cifratura) e l'altra autenticazione (firma). L'ordine può essere invertito, purché chi riceve sappia **prima cosa decifrare e con quale chiave**.

## Protocolli basilari di integrità

Finora abbiamo visto come ottenere:

- **Segretezza** → solo il destinatario può leggere il messaggio.
- **Autenticazione** → il destinatario sa con certezza chi ha mandato il messaggio.

Però ci manca ancora un pezzo fondamentale:

**Come faccio a sapere se il messaggio è stato modificato?**

Ed è qui che entra in gioco il concetto di **integrità**.

---

**Obiettivo:**

Garantire che **il messaggio ricevuto sia identico a quello inviato**, senza essere stato modificato da un attaccante.

**Soluzione sbagliata: solo hash (tipo  $h(m)$ )**

Viene spontaneo pensare: "Metto una funzione di hash come checksum!"

Tipo:  $A \rightarrow B : m, h(m)$

**MA ATTENZIONE!**

Un attaccante potrebbe tranquillamente:

- modificare  $m \rightarrow m'$
- calcolare  $h(m')$
- inviare  $m', h(m')$

...e sembrerebbe tutto a posto. Quindi: **non basta** un semplice hash.

---

**Soluzione giusta: usare l'hash insieme a una chiave**

**Caso 1 – Crittografia simmetrica**

Hai una chiave segreta condivisa  $K_{ab}$ .

- $A \rightarrow B : m, h(m, K_{ab})$

Questo è un **MAC** (Message Authentication Code)

Solo A e B possono calcolare o verificare l'hash, perché **solo loro conoscono**  $K_{ab}$ .

Un attaccante, non conoscendo  $K_{ab}$ , **non può manomettere** l'integrità del messaggio.

**Cos'è esattamente il MAC?**

Un **Message Authentication Code (MAC)** è un **meccanismo crittografico** che serve a **garantire l'integrità e l'autenticità** di un messaggio quando si usa **crittografia simmetrica**.

- **Cifratura:** attraverso una chiave trasforma il messaggio per **nasconderne il contenuto**.

- **MAC**: anche se utilizza comunque una chiave, serve solo a **verificare che il messaggio non sia stato modificato** e proviene da chi conosce quella chiave.

#### In parole semplici:

È una specie di **firma "leggera"** che permette al destinatario di sapere:

- se il **messaggio è stato modificato**
- se è stato **veramente inviato da chi possiede la chiave segreta**

#### Com'è fatto un *MAC*?

Un *MAC* è una **stringa (una sequenza di bit)** calcolata a partire da:

- **il messaggio  $m$**
- **una chiave segreta condivisa  $K$**

Il risultato è qualcosa tipo:

$$MAC = f(m, K)$$

Dove  $f$  è una **funzione crittografica**, spesso un **hash crittografico con chiave**, come HMAC (che usa SHA-256 o simili).

#### Cosa garantisce il *MAC*?

1. **Integrità**: Se qualcuno cambia anche un solo bit del messaggio, il *MAC* non torna più.
2. **Autenticazione**: Solo chi conosce la chiave  $K$  può calcolare un *MAC* valido.

#### Esempio pratico

**Supponiamo che A e B condividano una chiave segreta  $K$ .**

- A vuole inviare il messaggio  $m = "ciao"$  a B.
- A calcola  $MAC = HMAC(K, "ciao")$
- A invia:  $m = "ciao"$ ,  $MAC$

Quando B riceve il messaggio:

- Ricalcola  $HMAC(K, "ciao")$  → se coincide con quello ricevuto, ovvero *MAC*, allora **tutto ok**.
- Altrimenti, il messaggio è stato **manomesso o non proviene da A**.

#### Differenza con la Firma Digitale?

MAC	Firma Digitale
Usa crittografia simmetrica	Usa crittografia asimmetrica
Stessa chiave per A e B	Chiave privata per firmare, pubblica per verificare
Non garantisce la <b>non ripudiabilità</b>	Garantisce la <b>non ripudiabilità</b>
Più veloce	Più pesante (più lenta)

Un MAC è un "codice di autenticazione del messaggio" che protegge da modifiche e falsificazioni.

Si basa su una **chiave segreta condivisa**, e viene usato nei protocolli come IPSec, SSL/TLS, ecc.

#### Caso 2 – Crittografia asimmetrica

Qui si usa la **firma digitale**, cioè:

- $A \rightarrow B : m, sign_A(h(m))$
- oppure anche semplicemente:  $sign_A(m)$

Cioè: Alice **firma** il messaggio con la sua **chiave privata**  $k_a^{-1}$

Bob verifica con la **chiave pubblica**  $k_a$ .

**Cosa ottieni:**

- **SI Integrità:** se anche solo un bit cambia, la firma non è più valida.
- **SI Autenticazione:** solo chi ha la chiave privata di Alice poteva firmarlo.
- **NO segretezza:** il messaggio è in chiaro! Chiunque può leggerlo.

Integrità del messaggio  $m$  nella trasmissione da  $A$  a  $B$

- Crittografia simmetrica, omesso
- Crittografia asimmetrica
  - Prerequisito1:  $A$  abbia una chiave privata valida
  - Prerequisito2:  $B$  possa verificare che  $k_a$  è di  $A$

$$1. A \longrightarrow B : sign_A(m)$$

Stessi prerequisiti dell'autenticazione di  $A$  con  $B$

---

**Vuoi tutto insieme? (Segretezza + Integrità + Autenticazione)**

Allora devi **combinare i protocolli**, così:

**Caso crittografia asimmetrica:**

- $A \rightarrow B : sign_A(m_{k_b})$   
→ cioè: *prima* cifri  $m$  con la chiave pubblica di  $B$  ( $k_b$ ), *poi* **firmi tutto** con la tua chiave privata ( $k_a^{-1}$ ).

**Che cosa ottieni?**

- **Segretezza:** solo  $B$  può decifrare.
- **Integrità:** la firma protegge il contenuto.
- **Autenticazione:** la firma conferma l'identità del mittente.

Segretezza del messaggio  $m$  per  $A$  e  $B$ , autenticazione di  $A$  con  $B$ , integrità di  $m$  nella trasmissione da  $A$  a  $B$

- Crittografia simmetrica, omesso
- Crittografia asimmetrica
  - Prerequisito1:  $B$  abbia una chiave privata valida
  - Prerequisito2:  $A$  possa verificare che  $k_b$  è di  $B$
  - Prerequisito3:  $A$  abbia una chiave privata valida
  - Prerequisito4:  $B$  possa verificare che  $k_a$  è di  $A$

$$1. A \longrightarrow B : sign_A(m_{k_b})$$

Prerequisiti di segretezza e autenticazione qui sommati

---

**Nota sulla freshness**

I protocolli che abbiamo visto **non proteggono da attacchi di replay**, cioè:

| Un attaccante potrebbe prendere un messaggio valido vecchio e ritrasmetterlo.

Serve qualcosa in più, tipo:

- **timestamp**

- **nonce**

## Diffie-Hellman

Diffie-Hellman è un **protocollo di scambio di chiavi**: serve a permettere a due soggetti (es. Alice e Bob) di **condividere un segreto comune**, cioè una chiave simmetrica, **senza mai trasmetterla direttamente**.

### Come funziona (versione base, senza autenticazione)

1. Alice e Bob scelgono pubblicamente due parametri:

- $\alpha$  (un generatore)
- $\beta$  (un numero primo)

2. Alice genera un numero segreto casuale  $X_a$  e calcola:

- $Y_a = \alpha^{X_a} \text{ mod } \beta$
- Invia  $Y_a$  a Bob.

3. Bob fa lo stesso con  $X_b$  e calcola:

- $Y_b = \alpha^{X_b} \text{ mod } \beta$
- Invia  $Y_b$  ad Alice.

4. Poi:

- Alice calcola la chiave condivisa  $K_{ab} = Y_b^{X_a} \text{ mod } \beta$
- Bob calcola la stessa cosa  $K_{ab} = Y_a^{X_b} \text{ mod } \beta$

Grazie alle proprietà delle potenze modulari, entrambi ottengono **lo stesso identico valore**, che diventa la **chiave segreta**.

### A cosa serve il modulo?

L'**operazione modulo** serve per:

1. **Limitare i numeri** in un intervallo gestibile (es: 0 ...  $\beta-1$ )
2. **Garantire la sicurezza** grazie alle proprietà dell'aritmetica modulare
3. **Rendere difficile** (anzi, quasi impossibile) risalire all'esponente segreto → **problema del logaritmo discreto**

Chi osserva da fuori conosce solo  $\alpha$ ,  $\beta$ ,  $\alpha^{X_a} \text{ mod } \beta$ ,  $\alpha^{X_b} \text{ mod } \beta$ , e **non riesce a trovare**  $X_a$  o  $X_b$  facilmente.

### Ma se togliamo il mod $\beta$ ?

Se non usassimo il modulo:

- I numeri crescono **enormemente** (esponenziali!)
- Sarebbe **più facile** (in certi casi) risalire all'esponente
- **Non ci sarebbe il problema del logaritmo discreto**, che è la base della sicurezza di DH

### Il trucco di sicurezza: il logaritmo discreto

Dato  $\alpha$ ,  $\beta$  e  $Y_a = \alpha^{X_a} \text{ mod } \beta$ , trovare  $X_a$  è difficilissimo (è il problema del logaritmo discreto)

Questa **difficoltà matematica** nasce solo grazie all'aritmetica modulare.

Se non ci fosse il *mod*, l'attaccante potrebbe semplicemente fare il logaritmo classico.

### Il problema della sicurezza: man-in-the-middle

Il protocollo **non prevede autenticazione**. Questo significa che un attaccante (chiamiamolo Carlo) può:

- Intercettare il messaggio  $Y_a$  che Alice invia a Bob.
- Mandare a Bob un proprio valore  $Y_c$  spacciandosi per Alice.

- Fare lo stesso quando Bob manda  $Y_b$  ad Alice.

### Risultato?

- Alice e Bob **pensano** di parlare tra loro, ma **stanno scambiando la chiave con Carlo**.
- Carlo riesce così a stabilire **due chiavi distinte**, una con Alice e una con Bob, e può **intercettare e modificare i messaggi cifrati** → attacco man-in-the-middle riuscito!

### Come si risolve? Con l'autenticazione!

L'idea è: se posso **verificare che il messaggio arrivi proprio da Alice o da Bob**, evito il rischio di essere ingannato da un attaccante.

#### Due modi per aggiungere autenticazione:

##### **Metodo 1: Firmare il messaggio con la propria chiave privata**

- Alice invia  $\{Y_a\}_{K_a^{-1}}$  → firma digitale con la propria chiave privata
- Bob fa lo stesso con  $\{Y_b\}_{K_b^{-1}}$
- In questo modo, il destinatario può **verificare l'identità del mittente** grazie alla chiave pubblica.

##### **Metodo 2: Cifrare con la chiave pubblica del destinatario**

- Alice invia  $\{Y_a\}_{K_b}$  → solo Bob può leggere perché ha  $K_b^{-1}$
- Bob risponde con  $\{Y_b\}_{K_a}$

Questo garantisce **confidenzialità**, perché solo il vero destinatario può decifrare. Ma non garantisce da sola l'autenticazione, a meno che le chiavi siano verificate tramite certificati.

Aspetto	Pro	Contro
Sicurezza	Segretezza perfetta, chiave mai trasmessa	No autenticazione, rischio MitM
Efficienza	Rinnovo facile delle chiavi (forward secrecy)	Calcoli pesanti rispetto alla crittografia simmetrica
Applicazioni	Usato in TLS, SSH, IPSec	Parametri mal scelti = vulnerabilità

## RSA Key Exchange

È un algoritmo di crittografia asimmetrica, un metodo alternativo a Diffie-Hellman per scambiarsi **una chiave di sessione simmetrica**, ma usando la **crittografia asimmetrica**.

Anche se parliamo di crittografia asimmetrica, alla fine la comunicazione vera e propria avviene con crittografia simmetrica, perché è più veloce ed efficiente.

### Come funziona (step-by-step)

1. Bob ha una coppia di chiavi RSA:
  - Chiave pubblica  $K_b$
  - Chiave privata  $K_b^{-1}$
2. Alice vuole inviare a Bob una **chiave di sessione** simmetrica (es. per AES), chiamiamola  $K_s$ .
3. Alice fa:
  - $M = K_s K_b \rightarrow$  prende  $K_s$  e la **cifra con la chiave pubblica di Bob**
  - Invia  $M$  a Bob
4. Bob riceve  $M$  e fa:
  - $K_s = M^{K_b^{-1}} \rightarrow$  decifra il messaggio con la **sua chiave privata** e ottiene la chiave di sessione  $K_s$

Ora **entrambi condividono  $K_s$** , che possono usare per cifrare e decifrare i messaggi futuri con un algoritmo simmetrico.

### Cosa garantisce

- **Confidenzialità:** solo Bob può leggere  $K_s$ , perché solo lui ha  $K_b^{-1}$ .
- **Efficienza:** si usa RSA solo una volta, poi si passa a crittografia simmetrica (più veloce).

La crittografia simmetrica è più veloce perché usa operazioni semplici e ottimizzabili, mentre la crittografia asimmetrica è più pesante perché basata su problemi matematici complessi e costosi da calcolare.

### Cosa manca

- **Autenticazione:** Se Alice **non firma** niente con la sua chiave privata, **Bob non sa con certezza che è proprio lei** a mandare la chiave.  
L'attaccante potrebbe fingere Alice e inviare la propria chiave simmetrica cifrata a Bob, che ci casca.
- **Associazione chiave-proprietario:** Se non si è sicuri che la chiave pubblica appartenga davvero a Bob, un attaccante può intercettare il messaggio e sostituire la chiave pubblica con la sua.  
Così Alice finisce per cifrare  $K_s$  con la **chiave dell'attaccante**, e lui può leggerla!

Aspetto	Pro	Contro
Sicurezza	Cifratura con chiave pubblica, autenticazione possibile	Nessuna forward secrecy: se rubano la chiave privata, è finita
Prestazioni	Semplice da usare, immediato	Operazioni lente, inadatto per dati grandi
Implementazione	Niente parametri condivisi, facile da capire	Serve validazione della chiave pubblica

## Certificazione (crittografia asimmetrica)

Serve a **risolvere il secondo grande problema** della crittografia asimmetrica:

**Come faccio a sapere se una chiave pubblica appartiene veramente a una certa persona o server?**

La risposta è: **me lo certifica un'autorità fidata**.

### Cos'è un certificato digitale

Un **certificato digitale** è un documento elettronico che **lega un'identità** (es. il sito web [www.abc.it](http://www.abc.it)) a una **chiave pubblica**.

È fatto così:

{ Identità + Chiave Pubblica } firmato con la chiave privata della CA

Esempio:

$$\{ "A", "K_a" \}_{K_{CA}^{-1}}$$

- $A$  = identità (es. il nome del server o dell'azienda)
- $K_a$  = chiave pubblica
- $K_{CA}^{-1}$  = **firma digitale** dell'autorità di certificazione (CA)

Se riesco a **verificare la firma** (della CA) con la **chiave pubblica della CA**, allora **mi fido** che quella chiave pubblica sia davvero di "A".

**Devi verificare la firma della CA**, non la firma dell'identità (cioè di "A").

### Perché?

Quando ricevi un certificato come questo:

$$Sign_{K_{CA}^{-1}}("A", "K_a")$$

Tu:

1. **Prendi la chiave pubblica della CA**, che è **nota e fidata** (di solito preinstallata nel browser o sistema operativo).

2. **Verifichi la firma** usando questa chiave pubblica della CA ( $K_{CA}$ ).

3. Se la verifica va a buon fine, vuol dire che:

- La CA **ha davvero firmato** quella coppia (Identità + Chiave pubblica),
- Nessuno ha alterato il contenuto (perché la firma sarebbe invalida),
- Quindi puoi fidarti che " $K_A$ " è **davvero la chiave pubblica dell'identità "A"**.

Il software verifica se la firma sul certificato, decifrata con la chiave pubblica della CA, corrisponde ai dati dichiarati (identità e chiave pubblica): se sì, il certificato è valido e ci si può fidare. **Altrimenti verrà segnalato un errore di certificato non valido**, come:

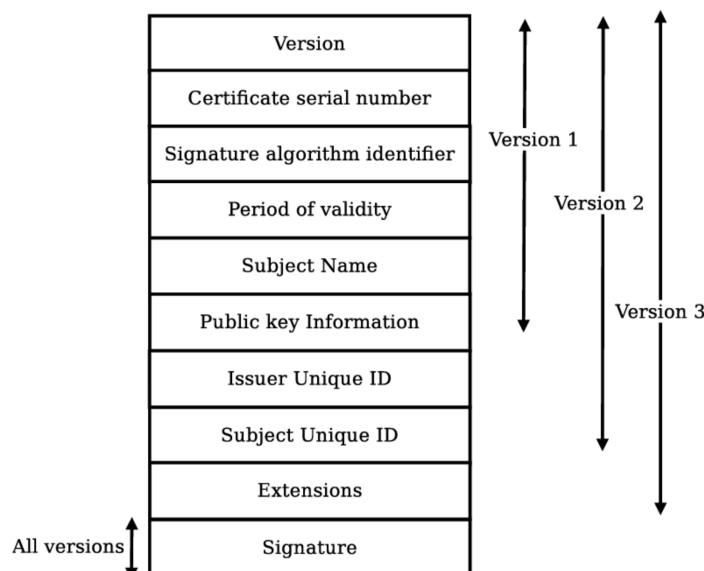
"Il certificato non è attendibile"

"Connessione non sicura"

#### Cosa non fai:

- Non puoi verificare una firma di "A" su sé stesso, perché **non ti fidi ancora di "A"** finché non verifichi il certificato!
- È proprio la **CA** che ti dice: "Sì, io garantisco che questa è la vera chiave pubblica di A".

#### Formato standard certificato X.509:



#### Come faccio a sapere se una CA è veramente affidabile?

Le CA (Certificate Authority) ritenute affidabili sono **preinstallate nei sistemi operativi e nei browser** dai loro sviluppatori (come Microsoft, Apple, Mozilla, Google...).

Questi produttori fanno un **processo di selezione molto rigoroso**: verificano che la CA abbia infrastrutture sicure, segua procedure certificate e abbia una reputazione affidabile. Solo così una CA entra nella lista delle "**trusted root CA**".

In pratica: ti fidi delle CA perché ti fidi di chi ha fatto il tuo browser o il tuo sistema operativo. E loro si prendono la responsabilità di fidarsi solo delle CA più serie.

#### Catena di fiducia & PKI

Ma come faccio a fidarmi della chiave pubblica della CA?

Serve una catena di fiducia, cioè una sequenza di certificati ognuno firmato da uno più "alto".

Esempio:

- Il certificato del sito è firmato da una **CA intermedia**
- La CA intermedia è firmata da una **RCA** (*root/primary certification authority*)
- La **RCA è auto-firmata** (cioè è il punto di partenza fidato)

Tutto questo è chiamato **PKI (Public-Key Infrastructure)**

È l'infrastruttura globale che gestisce chi si fida di chi.

La catena di fiducia di A è la lista completa dei certificati (fino a quello di root) che permettono, insieme alla chiave pubblica della RCA, di verificare il certificato di A

#### Come funziona nei browser

- I browser (Chrome, Firefox, ecc.) **hanno già al loro interno un set di certificati root** di cui si fidano.
- Quando apri un sito HTTPS, il browser **verifica tutta la catena di certificati** fino a una root.
- Se la catena è valida → Sito considerato sicuro  
Se no → Schermata rossa: "Connessione non privata"

#### Attenzione ai certificati self-signed

Un **certificato self-signed** è firmato da sé stesso, quindi **non ha una CA di riferimento**.

→ È come se io dicesse: "Fidati di me... perché lo dico io!"

Un attaccante potrebbe creare un certificato finto con scritto "Google.com" e firmarselo da solo. Il browser **non può sapere se è vero** e quindi ti avvisa del rischio.

**Bisogna accettare un certificato self-signed permanentemente o solo temporaneamente?**

Accettare un certificato self-signed **permanentemente** è rischioso perché manca una verifica da parte di una CA affidabile, rendendo possibile un attacco MITM in cui un attaccante si spaccia per il server e riceve dati cifrati; inoltre, il certificato resta nel sistema e viene accettato automaticamente in futuro, anche se è malevolo. Per questo motivo, è **preferibile accettarlo solo temporaneamente** e solo in contesti sicuri e controllati, come ambienti di sviluppo o test, dove si ha certezza dell'identità del server.

Tutti i **certificati root sono self-signed**, ma non tutti i **certificati self-signed** sono root (chiunque può generarsi un certificato self-signed, ma nessuno lo considererà attendibile se non è una RCA già nota al sistema).

In sostanza: **self-signed ≠ automaticamente affidabile**, a meno che non sia una **RCA già presente nella lista di fiducia del sistema**.

#### Certificati gratis vs a pagamento

- **Domain Validated (DV):**

Gratuiti, emessi facilmente (es. da Let's Encrypt), ma garantiscono solo che il dominio è sotto controllo del proprietario.

- **Extended Validation (EV):**

A pagamento, richiedono una **verifica approfondita** dell'identità legale, azienda, documenti, ecc.

#### Revoca dei certificati

A volte i certificati vanno **revocati**, prima della loro scadenza:

- Chiave privata compromessa o smarrita
- Uso fraudolento
- L'organizzazione non è più affidabile
- O cambio di Subject Identifier

Esistono due meccanismi per gestirlo:

- **CRL (Certificate Revocation List)** → una lista di certificati non più validi
  - La lista dei certificati revocati è firmata dalla CA che ha emesso i certificati ora revocati
  - Il Subject inoltra la richiesta
- **OCSP (Online Certificate Status Protocol)** → protocollo per controllare in tempo reale se un certificato è stato revocato, ovvero se si trova nella CRL

Il browser lo fa **automaticamente**, non bisogna quindi configurare nulla.

## Protocolli di sicurezza storici

### Contesto generale

Questi protocolli nascono per risolvere due esigenze fondamentali:

1. **Autenticazione** → essere sicuri dell'identità dell'interlocutore.
2. **Freshness** → essere sicuri che un messaggio sia "fresco", cioè non sia stato riutilizzato da un attaccante (replay attack).

| Replay Attack: Spacciare informazione (chiavi,...) obsoleta, magari violata, come recente.

---

### Needham-Schroeder Protocol (versione pubblica)

#### Obiettivo:

- Autenticazione reciproca
- Freshness
- (Opzionale) scambio di chiave

#### Fasi del protocollo:

1.  $A \rightarrow B : \{A, N_a\}_{K_B}$

| A manda il suo nome e una nonce ( $N_a$ ) a B, cifrata con la chiave pubblica di B.

2.  $B \rightarrow A : \{N_a, N_b\}_{K_A}$

| B risponde includendo la stessa nonce ( $N_a$ ) e una sua nuova nonce ( $N_b$ ), cifrate con la chiave pubblica di A.

3.  $A \rightarrow B : \{N_b\}_{K_B}$

| A rimanda indietro la nonce di B, cifrata con la chiave pubblica di B.

Con questa sequenza, B si fida che A sia A (perché è l'unica a poter decifrare il suo messaggio) e A si fida di B.

---

### Attacco di Lowe (man-in-the-middle)

Immagina che C sia l'attaccante.

1.  $A \rightarrow C : \{A, N_a\}_{K_C}$   
→ A crede di parlare con C.
2.  $C \rightarrow B : \{A, N_a\}_{K_B}$

→ C gira il messaggio a B fingendo di essere A.

3.  $B \rightarrow A : \{N_a, N_b\}_{K_A}$

→ B risponde ad A, che crede di parlare con C, ma riceve un messaggio da B.

4.  $A \rightarrow C : \{N_b\}_{K_C}$

→ A risponde pensando sia C il mittente del messaggio, ma in realtà è di B.

5.  $C \rightarrow B : \{N_b\}_{K_B}$

→ C invia la risposta finale a B, che ora è convinto di aver parlato con A.

**C ha impersonato A agli occhi di B.**

#### Soluzioni possibili (e non)

Vediamole con un commento rapido:

1.  $\{\{A, N_a\}_{K_A^{-1}}\}_{K_B}$  **NO**

→ Non basta: l'attaccante può fare replay verso B.

2.  $\{\{N_a, N_b\}_{K_B^{-1}}\}_{K_A}$  **SI**

→ Funziona: A riceve un messaggio firmato da B → autenticazione esplicita.

3.  $\{N_a, N_b, B\}_{K_A}$  **SI**

→ Funziona: A sa che il messaggio viene da B perché è indicato nel messaggio.

4.  $\{A, B, N_a\}_{K_B}$  **NO**

→ Cambia il primo messaggio, ma non risolve il problema.

Fix vero? Rendere chiaro chi ha inviato il messaggio → principio di esplicitazione del mittente. *Se le identità del mittente e del ricevente sono significative per il messaggio, allora è prudente menzionarle esplicitamente.*

#### Woo-Lam Protocol

##### Obiettivo:

- Autenticazione **unidirezionale**: A si autentica verso B
- Si usa una **TTP (Trusted Third Party)** che conosce le chiavi a lungo termine

##### Fasi:

1.  $A \rightarrow B : A$

2.  $B \rightarrow A : N_b$

(B chiede ad A di autenticarsi)

3.  $A \rightarrow B : \{N_b\}_{K_A}$

(A cifra la nonce di B con la sua chiave simmetrica)

4.  $B \rightarrow TTP : \{A, \{N_b\}_{K_A}\}_{K_B}$

(B chiede alla TTP se A ha davvero cifrato quella nonce)

5.  $TTP \rightarrow B : \{N_b\}_{K_B}$

(Se la TTP riesce a decifrare correttamente il messaggio con la chiave di A, allora conferma a B)

Se ci pensi, il TTP gioca un po' il ruolo di autorità che verifica "chi ha detto cosa".

Nel protocollo **Woo-Lam**, le chiavi a lungo termine come  $K_A, K_B$ , ecc. sono **condivise solo tra ciascun partecipante e la TTP (Trusted Third Party)**, quindi **non sono note agli altri partecipanti**.

Quindi:

- $K_A$ : chiave segreta condivisa **tra A e la TTP**

- $K_B$ : chiave segreta condivisa **tra B e la TTP**
  - Nessuno, tranne la TTP e il proprietario, può usare o conoscere quella chiave
- 

| B può decifrare  $\{N_b\}_{K_A}$  ?

**No, B non conosce  $K_A$ , quindi non può decifrare** un messaggio cifrato con quella chiave. Solo A e la TTP possono farlo.

---

Questo è un principio chiave del Woo-Lam (e di molti protocolli simili come Needham-Schroeder):

La TTP è l'unica entità che **conosce tutte le chiavi a lungo termine** e ha il compito di generare e distribuire chiavi di sessione, servendo da ponte sicuro tra le parti.

---

#### Attacco di sessione parallela

Immagina che C voglia fingersi A, mentre A è offline:

1.  $C \rightarrow B : A$

C, che finge di essere A, manda il messaggio a B dicendo di essere A.

2.  $B \rightarrow A : N_b$

B risponde ad A (C lo intercetta) indicando la *nonce*  $N_b$  cifrata con la propria chiave.

- all'utente A è associata la *nonce*  $N_b$

3.  $C \rightarrow B : \{N_b\}_{K_C}$

Secondo protocollo, C (che impersona A) dovrebbe mandare a B la stessa *nonce*  $N_b$  cifrata con la chiave di A, ma per creare confusione, cifra in modo errato con la chiave di C.

- C (che impersona A) invia la *nonce* esatta cifrando con una chiave errata

4.  $B \rightarrow TTP : \{A, \{N_b\}_{K_C}\}_{K_B}$

B invia al TTP ciò che ha ricevuto da A (ossia da C).

5.  $TTP \rightarrow B : \{N''_b\}_{K_B}$

Quando il TTP riceve questo messaggio da B lo decodifica, in quanto possiede tutte le chiavi. Quando apre il messaggio, il TTP si rende conto che il mittente è A, quindi apre ciò con la chiave di A. Ma dato che era codificato con la chiave di C, quello che si otterrà alla fine è un numero sbagliato, perciò una *nonce* non giusta, chiamata  $N''_b$

MA...

C può anche contemporaneamente fare:

1.1  $C \rightarrow B : C$

C questa volta dice la verità, inviando correttamente il messaggio secondo protocollo.

2.1  $B \rightarrow C : N'_b$

B risponde a C indicando la *nonce*  $N'_b$  cifrata con la propria chiave.

- all'utente C è associata la *nonce*  $N'_b$

3.1  $C \rightarrow B : \{N_b\}_{K_C}$

Secondo protocollo, C dovrebbe mandare a B la stessa *nonce*  $N'_b$  cifrata con la chiave di C, ma per creare confusione, utilizza erroneamente la *nonce*  $N_b$ .

- C invia la *nonce* sbagliata cifrando con una chiave giusta

4.1  $B \rightarrow TTP : \{C, \{N_b\}_{K_C}\}_{K_B}$

B invia al TTP ciò che ha ricevuto da C.

5.1  $TTP \rightarrow B : \{N_b\}_{K_B}$  (ACCETTATA)

Quando il TTP riceve questo messaggio da B, lo decodifica, lo apre esattamente in quanto mittente e chiave di codifica sono esatti, e ritorna la *nonce*  $N_b$

**B, quando riceve dal TTP le due *nonce* capisce che:**

- la *nonce*  $N''_b$  è sbagliata, perciò la scarta;
- la *nonce*  $N_b$  è giusta, e quindi l'associa all'utente A (che in realtà è C).

Quindi, ogni messaggio successivo tra C e B risulta valido, solamente che B crede di comunicare con A, ma ciò non è vero.

L'attacco della **sessione parallela** funziona perché:

**B non distingue tra due conversazioni in parallelo**, e accetta una risposta  $\{N_b\}_{K_C}$  credendola da A, perché la manda alla TTP assieme al nome "A".

La TTP decifra  $\{N_b\}_{K_C}$  con la chiave **di A**, ma il messaggio è stato cifrato da **C**, quindi la decifrazione fallisce (ma la TTP non se ne accorge, o restituisce comunque qualcosa).

L'attacco sfrutta il fatto che **il contenuto cifrato non è legato esplicitamente all'identità del mittente**, quindi si può costruire un messaggio falso con chiavi sbagliate e ingannare il protocollo.

Serve quindi esplicitare sempre chi ha detto cosa, legandolo ai messaggi con cripatura o firma.

### Fix proposti

1.  $A \rightarrow B : \{A, N_b\}_{K_A}$  **NO**  
→ Attaccante lo costruisce, non ci possiamo fidare.
2.  $TTP \rightarrow B : \{A, N_b\}_{K_B}$  **SI**  
→ TTP specifica chiaramente chi è il proprietario della chiave usata → esplicitazione del mittente → fix corretto.
3.  $B \rightarrow TTP : \{A, \{A, N_b\}_{K_A}\}_{K_B}$  **NO**  
→ Non serve: TTP comunque guarda solo la chiave usata.
4.  $B \rightarrow A : N_b$ ,  $B$  **NO**  
→ Non migliora la sicurezza.

## Sicurezza sui vari livelli OSI

L'idea è questa: possiamo mettere la sicurezza a **qualsiasi livello** dello stack TCP/IP, ma ogni scelta ha pro e contro.

### 1. Livello Applicazione

- Esempi: HTTPS, PGP (email), S/MIME, autenticazioni in app, ecc.
- **Pro:**
  - Massima flessibilità: ogni app decide cosa proteggere e come.
  - Non richiede modifiche a OS o protocollo.
- **Contro:**
  - È facile sbagliare: la sicurezza è demandata agli sviluppatori delle app.
  - Le app possono "ignorare" il contesto di rete → possibili errori concettuali.
  - Ogni app deve reinventare la ruota.

### 2. Livello Trasporto

- Esempi: SSL/TLS, DTLS.
- **Pro:**
  - Le app non devono più preoccuparsi della cripatura: "ci pensa il trasporto".
  - È un buon compromesso tra sicurezza e trasparenza.
- **Contro:**
  - Bisogna comunque configurare le app perché usino questi protocolli.
  - Non protegge i pacchetti a livello IP.

### 3. Livello Rete

- Esempi: IPSec.
- **Pro:**
  - Massima trasparenza: le app nemmeno si accorgono che c'è la sicurezza.
  - Protegge tutto il traffico, a prescindere dall'app.
- **Contro:**
  - Più difficile da configurare.
  - Rischia di appesantire il traffico (es. cifratura di tutto).
  - Può richiedere modifiche al SO o supporto hardware.

Nota importante: più si scende di livello, più la sicurezza è trasparente, ma meno è flessibile.

La scelta del livello in cui inserire la sicurezza è una decisione architettonica importante:

- Vuoi che tutte le app siano protette senza modificare nulla? Vai in basso (es. IPSec).
- Vuoi flessibilità e controllo app per app? Vai in alto (es. HTTPS, PGP).
- Vuoi un buon equilibrio? Vai al livello trasporto (es. TLS).

### Chaffing & Winnowing

Questo è un metodo **non convenzionale** per ottenere riservatezza **senza cifrare il messaggio**, ma sfruttando la **non distinguibilità**.

#### Il nome

- **Chaff** = pula, ovvero i dati falsi.
- **Winnowing** = vagliatura: è l'azione di separare il grano (buono) dalla pula (inutile).
- In informatica significa "mescolare messaggi veri e falsi in modo che solo il destinatario possa distinguere quelli veri".

#### Come funziona?

1. **S (mittente)** e **R (ricevente)** si accordano su una chiave segreta  $k$  tramite, per esempio, **Diffie-Hellman**.
2. S manda (in chiaro):
  - **Messaggio reale:**  $m$ , assieme al  $MAC(m, k)$  → una sorta di "firma" verificabile solo con la chiave  $k$ .
  - **Messaggi fasulli:** coppie  $x, y$  dove  $x$  è un messaggio casuale e  $y$  è un  $MAC$  sbagliato (generato a caso o su un altro messaggio).
3. **R (il ricevente)** riceve **tutti** i messaggi (veri e falsi), ma è l'unico che può calcolare  $MAC(m, k)$  e confrontarlo con quello ricevuto per sapere **quale messaggio è autentico**.
  - a. Quindi la **riservatezza è ottenuta non perché il messaggio è nascosto**, ma perché **solo il destinatario sa quali messaggi sono autentici**.

L'attaccante non sa quale sia il messaggio corretto perché:

- I dati non sono cifrati, ma
- Non è in grado di distinguere un  $MAC$  valido da uno falso **senza conoscere  $k$** .

### Proprietà di sicurezza ottenute

- **Riservatezza:** l'attaccante non sa quale messaggio è autentico.
- **Integrità:** se modifica il messaggio, il  $MAC$  non combacia più.
- **Autenticità:** solo chi conosce la chiave  $k$  può costruire un  $MAC$  valido.

Quindi, otteniamo tutte le proprietà che normalmente otteniamo con la crittografia... **ma senza cifrare!**

### Esempio concreto

Supponi che il mittente voglia inviare:

Messaggio: "Il bersaglio è Roma"

Senza cifratura, l'attaccante legge subito il messaggio.

Con chaffing & winnowing:

- L'attaccante intercetta 100 messaggi diversi (tipo: "ciao", "test", "il bersaglio è roma", "banana", ecc.)
- Solo uno ha il MAC corretto.
- Solo il destinatario sa quale è giusto.
- L'attaccante **non può sapere quale è quello vero**.

### Limiti del Chaffing & Winnowing

- **Non è scalabile:** per ogni messaggio vero bisogna generare tanti messaggi fasulli → traffico di rete esplode.
- **Non è standardizzato né ampiamente usato:** resta più che altro un'idea interessante per dimostrare che la sicurezza può derivare anche da **non distinguibilità** (come nella steganografia).

### Steganografia vs Chaffing & Winnowing

Tecnica	Idea principale	Come garantisce la sicurezza
Steganografia	Nascondere il messaggio vero in uno innocuo	L'attaccante non sa che c'è un messaggio
Chaffing & Winnowing	Mischiare messaggi veri e falsi	L'attaccante non sa quale è quello vero

Entrambe puntano sulla **non distinguibilità** anziché sulla cifratura per proteggere l'informazione.

## IPSec

**IPSec** (Internet Protocol Security) è un insieme di protocolli che fornisce **sicurezza a livello IP**. Quindi, mentre altri protocolli lavorano a livelli superiori (tipo TLS che sta sopra TCP), IPSec protegge **direttamente i pacchetti IP**.

Ogni pacchetto IP può essere autenticato, cifrato o entrambi.

### I componenti principali di IPSec

#### 1. AH (Authentication Header)

Serve per **autenticare** e garantire l'**integrità** del pacchetto IP.

Protegge da modifiche o spoofing.

Non cifra i dati (niente confidenzialità).

#### 2. ESP (Encapsulating Security Payload)

Serve per **cifrare il contenuto** (payload) del pacchetto e, optionalmente, autenticarlo.

Fornisce confidenzialità + integrità + autenticazione (optional).

#### 3. IKE (Internet Key Exchange)

Serve per **negoziare e scambiare le chiavi** tra due nodi che vogliono usare IPSec.

Usa **Diffie-Hellman** (una sua variante: Oakley) per creare una chiave simmetrica condivisa.

Definisce anche le **Security Association (SA)**.

### Cos'è una Security Association (SA)?

Una SA è un **accordo unidirezionale** tra due nodi su **quale sicurezza applicare**.

Esiste una SA per ogni direzione (es: A→B ha una SA, B→A ne ha un'altra).

Ogni SA contiene:

1. **SPI (Security Parameter Index)** – un identificatore unico per quella SA
2. **Indirizzo IP di destinazione** – perché la SA è per comunicazioni unicast
3. **Protocollo di sicurezza usato (AH o ESP)**
4. **AH info / ESP info** - informazioni aggiuntive sull'algoritmo di autenticazione / codifica
5. **Lifetime** – tempo di validità della SA

Le SA sono salvate nel **Security Association Database (SAD)**.

#### Come funziona IPSec in pratica?

Prima di tutto i due nodi usano **IKE** per scambiarsi le chiavi e stabilire le SA.

IKE consta di due protocolli:

1. Oakley: livello applicazione, variante di Diffie-Hellman per scambiare la chiave iniziale.
2. ISAKMP: livello trasporto, per generare le chiavi di sessione a partire dalla chiave iniziale.

Dopo, ogni pacchetto IP può essere protetto in due **modalità**:

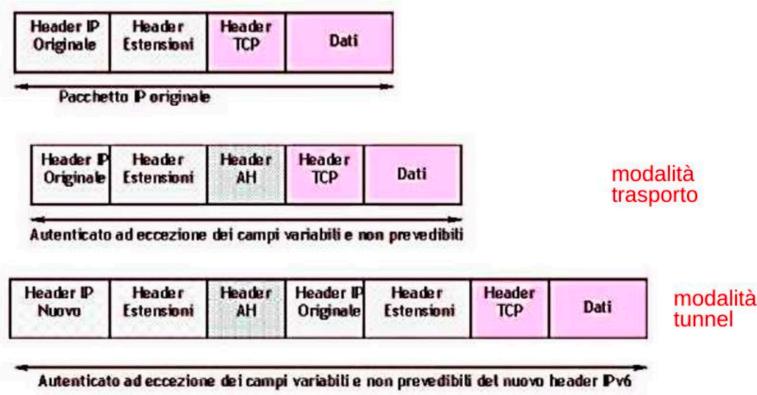
#### Modalità Trasporto

- Protegge **solo il payload del pacchetto IP**
- L'header IP originale rimane **visibile**
- Usata in comunicazioni **end-to-end** (es. da PC a PC)
- Richiede che **entrambi i nodi supportino IPSec**

#### Modalità Tunnel

- Protegge **l'intero pacchetto IP originale** (dati + header)
- Il pacchetto originale viene **incapsulato** in un nuovo pacchetto IP
- Usata per VPN, **autenticazione tra due gateway/router/firewall**, es. tra due sedi aziendali.
- I due host non devono per forza conoscere IPSec: la protezione è **intermedia**.

È utile, ma **non è Zero Trust**, perché l'host finale non verifica nulla: si fida del router.



#### AH: Authentication Header

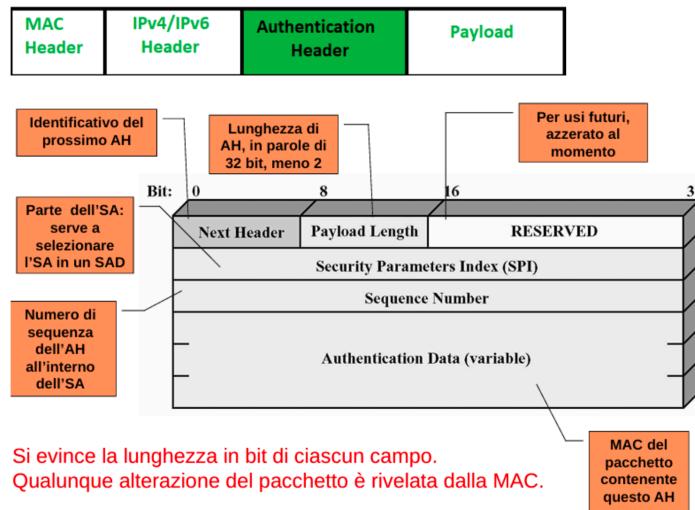
- Aggiunge un'intestazione al pacchetto IP
- **Non cifra**, ma garantisce che il pacchetto non sia stato modificato
- Calcola un **MAC (Message Authentication Code)** usando una chiave segreta condivisa

#### Protegge:

- Il payload
- I campi IP **non modificabili** (es. indirizzo sorgente e destinazione)

- **NON protegge** i campi che possono cambiare in transito (es. TTL, checksum IP)

Previene Replay Attack:



Un attaccante può **intercettare un pacchetto valido e rilanciarlo più volte** verso il destinatario.

Senza contromisure, il destinatario non avrebbe modo di sapere che quel pacchetto è **duplicato**.

#### La soluzione: la finestra scorrevole (sliding window)

Ogni pacchetto ha:

- Un **numero di sequenza** incrementale (es. 1, 2, 3, ...)
- Un **MAC (Message Authentication Code)** per verificarne l'autenticità

Funzionamento della finestra:

- La finestra ha dimensione **W** (es. 64 pacchetti).
- Tieni **traccia dei pacchetti ricevuti correttamente** (in una bitmap).
- Il ricevente mantiene **N**, il massimo numero di sequenza **ricevuto e accettato finora**.

**Cosa succede alla ricezione di un nuovo pacchetto con numero  $M$ ?**

**1. Se  $M > N$ :**

- È un pacchetto nuovo, fuori dalla finestra.
- Se il **MAC** è valido, viene **accettato**.
- La finestra viene **fatta scorrere verso destra** fino a includere M.
- Si aggiornano N e la bitmap.

**2. Se  $M$  è dentro la finestra ( $N - W < M \leq N$ ):**

- Si controlla la bitmap:
  - Se il pacchetto **non è già stato ricevuto** e il MAC è valido → **accettato**.
  - Se **è già stato ricevuto** → **scartato** (è un replay).

**3. Se  $M \leq N - W$ :**

- Il pacchetto è **troppo vecchio, fuori dalla finestra a sinistra**.
- **Scartato** direttamente → potenziale replay.

**4. Se il **MAC** fallisce:**

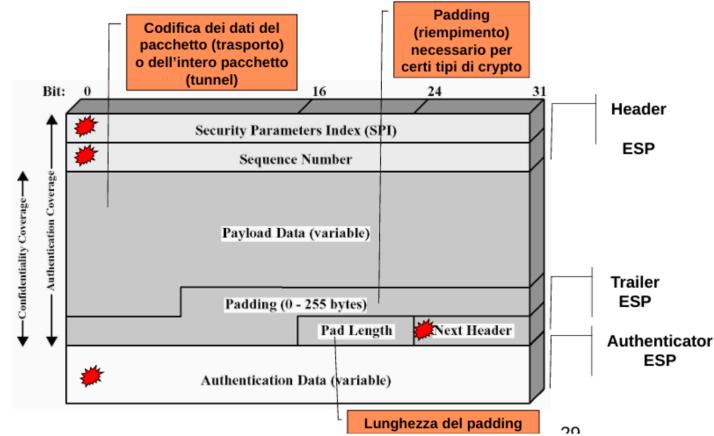
- **Scartato**, indipendentemente dal numero di sequenza.

### Esempio concreto (con $W = 64$ , $N = 100$ )

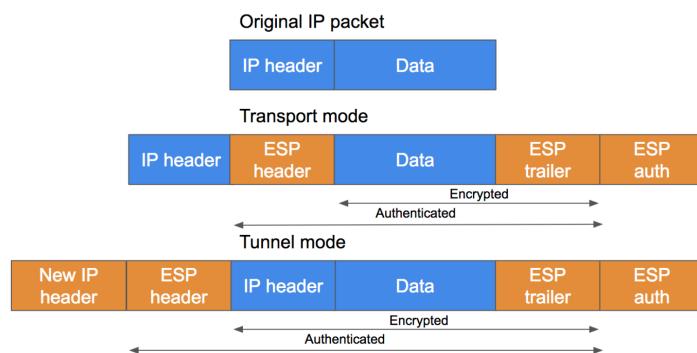
- Pacchetto con  $M = 105 \rightarrow$  nuovo  $\rightarrow$  accettato  $\rightarrow$  finestra si sposta a destra (fino a 105).
- Pacchetto con  $M = 97 \rightarrow$  dentro finestra  $\rightarrow$  se non visto prima  $\rightarrow$  accettato.
- Pacchetto con  $M = 50 \rightarrow$  troppo vecchio  $\rightarrow$  **scartato**.

### ESP: Encapsulating Security Payload

- Cifra il **payload** del pacchetto (dati veri e propri)



- Può **autenticare** (MAC), ma è opzionale
- Supporta modalità **trasporto** e **tunnel**

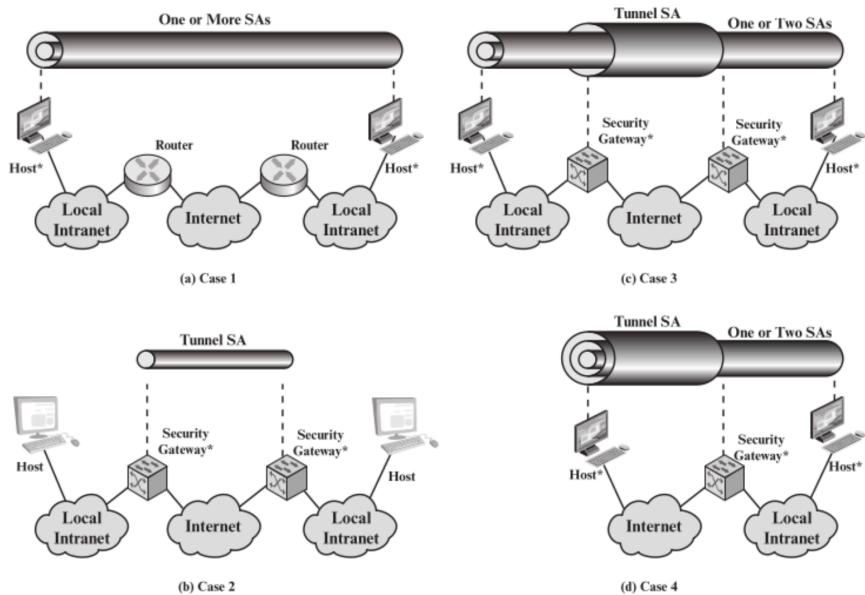


In tunnel, nasconde anche l'indirizzo IP reale del destinatario

(simile al comportamento di Tor, anche se lì si usano più livelli di cifratura)

### Esempi di combinazioni SA

1. **AH trasporto + ESP trasporto**
  - Protezione end-to-end (tra host A e host B)
2. **AH tunnel + ESP tunnel**
  - Protezione gateway-to-gateway (tra firewall o router)
3. **Mix punto a punto + tunnel**
  - Protezione da host a router (es. AH), poi tunnel tra gateway, poi ancora trasporto tra router e host
4. **ESP in tunnel fino al gateway, poi ESP trasporto fino al destinatario finale**
  - Per esempio: proteggi tutto tra A e firewall, poi solo payload fino a B



### Retrocompatibilità

Se un nodo riceve un pacchetto con un **SPI non noto** (cioè non ha la SA associata), il pacchetto può essere trattato **come un normale pacchetto IP**, cioè passa "in chiaro".

Questo lo rende compatibile con sistemi che **non usano IPSec**.

### Potenziali semplificazioni di IPSec

1. Eliminare AH: usare sempre ESP con autenticazione
2. Eliminare modalità trasporto e usare sempre solo tunnel
  - a. IPSec dovrebbe essere su ogni nodo
  - b. Un mondo di VPN
3. Ridefinire le SA come associazioni bidirezionali

## Intrusion Detection

È l'insieme di tecniche e strumenti che servono per **rilevare attività malevole all'interno di un sistema, da parte di un utente** (non di un semplice virus!).

Quindi:

"C'è qualcuno nel sistema che non dovrebbe esserci... o si sta comportando in modo sospetto".

### Intrusione ≠ Malware

Una **intrusione** implica un **attore umano** o comunque **attivo** che:

- entra nel sistema **abusando di vulnerabilità**,
- oppure **aumenta i propri privilegi**,
- e può poi **installare malware** (che è solo uno **strumento** dell'attacco).

### Intrusion Detection System (IDS)

Un **IDS** ha l'obiettivo di **identificare** attività sospette, non di prevenirle. Se vuoi prevenzione, si parla di **IPS** (Intrusion Prevention System).

#### Esempio pratico:

- Un utente normale lancia un paio di shell: tutto ok.

- Un intruso lancia **10 shell contemporaneamente** per fare privilege escalation → comportamento **anomalo**, potenzialmente segnalato dall'IDS.

#### Come distinguere i comportamenti "lecati" da quelli "alieni"?

Eh... questa è la parte difficile!

#### Problema:

- Alcuni comportamenti **lecati ma inusuali** (es: 10 terminali aperti) sembrano **sospetti**.
- Non esiste un algoritmo deterministico che ci dica "questo è malevolo" con certezza.

#### Soluzione:

Si usano **tecniche di IA e machine learning**.

#### Ma attenzione: queste tecniche non sono perfette!

- Possono dare:
  - **Falsi positivi** (alert per comportamenti lecati),
  - **Falsi negativi** (non segnalano un vero attacco).

#### La qualità dell'IDS si misura in base a:

- **Percentuale di falsi positivi**,
- **Percentuale di falsi negativi**,
- **Tempo di risposta**,
- **Affidabilità nel tempo**.

#### Logging: il cuore dell'IDS

Il logging tiene traccia di tutto: chi ha fatto cosa, quando, come e con quali risorse.

#### Denning e i suoi record:

La scienziata Dorothy Denning propose un modello di logging per l'intrusion detection.

Ogni evento loggato ha **6 campi** fondamentali:

1. **Soggetto** → chi ha eseguito l'azione (utente, processo).
2. **Azione** → cosa ha fatto (es. lettura, scrittura, login...).
3. **Oggetto** → su cosa ha agito (file, cartella, rete...).
4. **Eccezione** → se ci sono stati errori (es. accesso negato).
5. **Uso di risorse** → CPU, RAM, disco ecc.
6. **Timestamp** → quando è accaduto.

### Tecniche di Intrusion Detection

#### 1. Tecniche Assolute

- Si basano su **eventi specifici**, indipendentemente dal contesto passato.
- Es: 3 tentativi di login falliti consecutivi → blocco account.

#### 2. Tecniche Dipendenti dal passato

- Si basano su **modelli statistici** di comportamento.
- Es: se un utente di solito accede al sistema 1 volta al giorno, e oggi lo fa 20 volte → anomalia!

### Due tipi di modelli di rilevamento

#### Rilevamento statistico

- Si costruisce un **profilo dell'utente normale**.
- Se un comportamento **si discosta troppo**, scatta l'allarme.
- Usa dati tipo:

- Media e deviazione standard (quanto si discosta),
- Serie temporali (quanto spesso succede),
- Pattern operativi.

#### Rilevamento a regole

- Basato su **policy definite**.
- Es: "nessun utente normale può accedere alla cartella /root", oppure "solo admin può installare software".

## Denial of Service (DoS)

È un attacco che **satura un servizio**, rendendolo inutilizzabile per gli utenti legittimi.

#### Tre tipi di DoS:

1. **cDoS**: Consumo di risorse computazionali (CPU).
2. **mDoS**: Consumo della memoria.
3. **bDoS**: Consumo della banda (più comune).

#### Esempio:

Un attaccante invia **migliaia di richieste** a un server → il server è occupato e **non risponde più agli utenti reali**.

#### DDoS: Distributed DoS

Stesso concetto, ma con **più macchine** attaccanti → più difficile da fermare.

Es: una botnet controllata da un attaccante invia richieste contemporaneamente a un server.

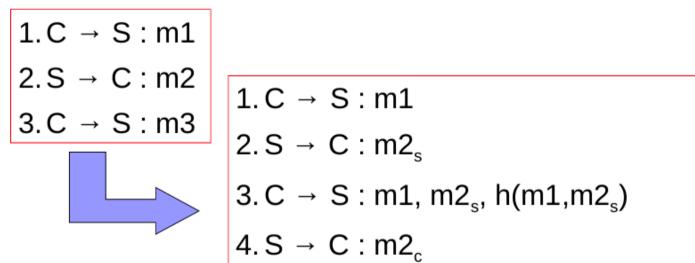
#### Come ci si difende?

##### 1. Code:

- Le richieste vengono messe in coda.
- Funziona, ma se la coda è piena... qualcuno rimane comunque fuori.

##### 2. Cookie Transformation:

- Il server **invia prima un cookie/captcha**, per verificare se il client è legittimo.
- Solo dopo invia la risposta completa.
- Serve a bloccare **loop automatici** o bot.

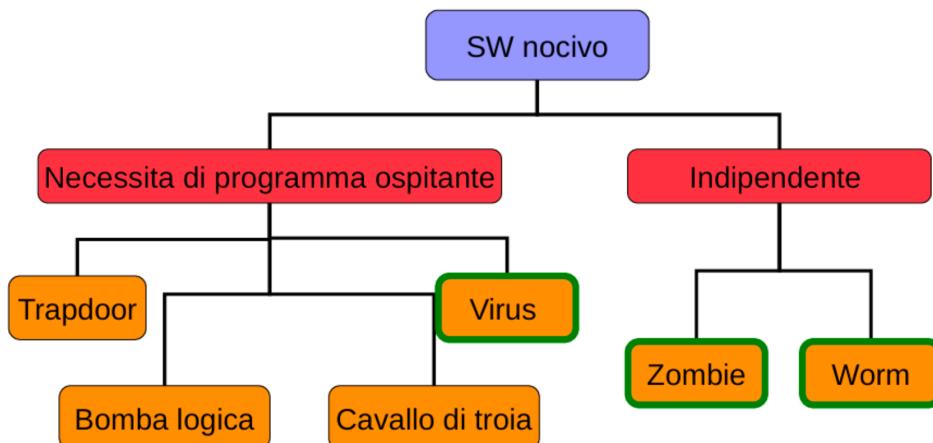


#### Differenze tra DoS e Intrusione

DoS	Intrusioni
<ul style="list-style-type: none"> <li>■ Effetti temporanei</li> <li>■ Immediatamente pubblico</li> <li>■ Blocco delle risorse</li> </ul>	<ul style="list-style-type: none"> <li>■ Effetti generalm. permanenti</li> <li>■ Spesso non reso pubblico</li> <li>■ Uso illecito di risorse</li> </ul>

Parametro	Modello	Tipo di intrusione rilevata
Frequenza di login, in base a giorno e ora	Media e deviazione standard	Tentativi di login ad orari non abituali possono rappresentare un'intrusione
Frequenza di login, in base alla posizione / località	Media e deviazione standard	Intrusione da una particolare località mai registrata o da cui l'utente raramente ha effettuato l'accesso
Tempo trascorso dall'ultimo login	A soglia	Effettuare l'accesso ad un account che non veniva acceduto da tempo potrebbe indicare un'intrusione
Quantità dell'output rispetto alla posizione	Media e deviazione standard	Un'eccessiva quantità di dati da una località remota può significare che c'è stato un leak di dati sensibili
Utilizzo delle risorse	Media e deviazione standard	Un utilizzo anomalo o eccessivo delle risorse (computazione ed I/O) può essere indicativo di un'intrusione
Numero di fallimenti durante il login	A soglia	Tentativo di accesso con guessing
Numero di fallimenti durante il login da un applicativo specifico	A soglia	Tentativo di accesso

## Tassonomia di software nocivi



### Cos'è un malware?

La parola **malware** viene da “**malicious software**” e indica **qualsiasi software progettato per danneggiare, infettare o compromettere un sistema**.

Sotto questa categoria rientrano diversi tipi, ciascuno con comportamenti e scopi diversi.

### Backdoor / Trapdoor

Una **backdoor** è un accesso nascosto inserito intenzionalmente in un sistema per permettere a un utente (legittimo o malevolo) di bypassare i normali controlli di sicurezza, come autenticazione o firewall. Una **trapdoor**, invece, è un termine più generico che indica una funzionalità segreta o una vulnerabilità lasciata apposta nel software, spesso non documentata, che può o meno fornire accesso remoto. Le due possono coincidere, ma **non sempre**: tutte le backdoor sono trapdoor, ma non tutte le trapdoor sono vere backdoor. Né la backdoor né la trapdoor sono di per sé **malware**, perché non sono necessariamente progettate per causare danni o infettare un sistema autonomamente; diventano strumenti di attacco solo **quando vengono sfruttate da un attaccante** o inserite con intento malevolo.

## Logic Bomb

Un pezzo di codice che **rimane dormiente** finché **non si verifica una certa condizione**, poi si attiva.

- Attiva un'azione distruttiva (es. cancella file, criptaggio ecc.)
- Le condizioni possono essere: una data, l'apertura di un file, o l'assenza di un file specifico

Esempio: "se oggi è il 1° aprile → formatta il disco"

---

## Trojan Horse (Cavalo di Troia)

Un software **apparentemente innocuo** che contiene codice malevolo al suo interno.

- L'utente lo installa pensando che sia legittimo (es. gioco, utility...)
- In realtà, dietro le quinte, **viola la sicurezza del sistema**

Esempio: un'app che dice di essere un convertitore PDF, ma installa un keylogger.

---

## Zombie

Un dispositivo **infettato e controllato a distanza**, usato **per scopi malevoli** (di solito da remoto, e all'insaputa dell'utente).

- Fa parte di una **botnet** (rete di zombie)
- Usati per **attacchi DDoS**, spam, mining ecc.

Esempio: il tuo PC è infetto e partecipa a un attacco DDoS senza che tu lo sappia.

---

## Worm

Un **worm** è un tipo di malware **indipendente, autonomo**, cioè si **propaga da solo** da un computer all'altro sfruttando vulnerabilità nei sistemi.

A differenza dei virus, **non ha bisogno di un file ospitante** da infettare: è autonomo e spesso molto veloce.

- Estremamente contagioso
- Può portare altri malware con sé

### Caso studio: Morris Worm (1988)

#### Chi era Morris?

Robert Tappan Morris era uno studente del MIT nel 1988, figlio di un ingegnere molto noto che lavorava nei laboratori Bell.

Era appassionato di reti e voleva **capire quanti computer erano connessi a Internet**, che all'epoca era ancora ARPANET, una rete accademica e militare molto limitata (circa 60.000 computer).

---

#### Obiettivo di Morris

L'intenzione di Morris **non era malevola**. Voleva solo **contare quanti computer erano connessi a Internet**.

Ma per farlo, scrisse un programma che **si replicava automaticamente...** e le cose **gli sono un po' sfuggite di mano**

---

#### Worm di Morris – Propagazione

##### 1. Come entra in una macchina remota?

Il worm si propaga sfruttando **3 tecniche diverse**, tutte alternative:

###### a. Attacco "known-ciphertext" sul file delle password

- Unix ha un file chiamato `/etc/passwd` **leggibile da tutti** (errore di visibilità).
- Le password erano cifrate, ma era **nota la funzione di hash** (`crypt()`).
- Questo è un attacco di tipo *known-ciphertext*, perché:
  - Conosci la funzione di cifratura.
  - Conosci l'output cifrato (l'hash).
  - Provi a **indovinare l'input** (la password).

###### Cosa faceva Morris?

1. Prendeva il **nome utente** (es. `emanuel`) e provava sue **permutazioni** (es. `emanuel1`, `emanuel123`...).
  2. Se non funzionava, provava con **una lista di 432 parole** predefinite (parole comuni, candidati statistici).
  3. Se ancora niente, provava **tutte le parole** nella directory `/local` (dizionario di sistema).
  4. Se riusciva a trovare la password di un utente, faceva login da remoto e scaricava il suo codice.
- 

#### b. BOF (Buffer Overflow) tramite `fingerd`

- `finger` è un comando che chiede info su un utente remoto.
- Il demone `fingerd` riceve questa richiesta e la elabora.

#### Il problema:

- `fingerd` aveva un **buffer** troppo piccolo per i dati in input.
- Morris gli mandava un input **più lungo del buffer** → **overflow** della memoria.

#### Cosa succedeva:

- Il buffer traboccava e sovrascriveva l'**indirizzo di ritorno**.
  - Il worm metteva **una shellcode** nel buffer.
  - Quando `fingerd` cercava di uscire, **saltava alla shellcode**, che avviava il download delle **99 righe di codice**.
- 

#### c. Trapdoor in `sendmail`

- `sendmail` è un programma per spedire email.
- In ascolto continuo su internet, **poteva essere messo in modalità di debug**.

#### La trapdoor:

- In modalità debug, `sendmail` eseguiva **comandi arbitrari** ricevuti via rete (!).
  - Morris usava questa trapdoor per dire a `sendmail` :  
“Esegui questi comandi per scaricare e compilare il worm”.
- 

### 2. Scaricamento e propagazione

- Una volta entrato nella macchina (con **una delle tre tecniche sopra**), il worm:
    1. **Scaricava 99 righe di codice C.**
    2. Le compilava localmente (quindi il worm era **portabile**, non binario fisso).
    3. Il programma risultante scaricava e installava **l'intero worm**.
    4. E poi ricominciava: cercava altre macchine sulla rete da infettare.
- 

#### Perché è diventato un disastro?

Il problema era proprio questo comportamento:

**invece di infettare ogni macchina una sola volta**, continuava a farlo più volte, anche su computer già infetti.

Questo ha causato:

- **Sovraccarico delle CPU** (macchine lente o inutilizzabili)
  - **Blocchi di rete**
  - **Fino al 10% di ARPANET giù** per qualche giorno (e nel 1988 era tantissimo!)
- 

#### Cosa è successo a Morris?

Anche se non voleva creare danni, Morris è stato:

- Il **primo condannato** secondo il **Computer Fraud and Abuse Act** del 1986
  - Condannato a **3 anni di libertà vigilata, 400 ore di lavori socialmente utili e 10.000 dollari di multa**
  - In seguito è diventato professore di informatica alla Cornell University!
- 

#### Perché è importante questo worm?

Il worm di Morris ha:

- **Inaugurato l'era degli attacchi automatici via rete**
  - Mostrato quanto siano pericolosi i sistemi non aggiornati
  - Fatto nascere i primi **CERT** (Computer Emergency Response Teams)
  - Insegnato al mondo che **anche una buona intenzione può causare un disastro**, se non si capisce bene cosa si sta facendo
- 

## Virus

Un software nocivo che **si attacca ad altri programmi o file eseguibili**, **si propaga** quando questi vengono eseguiti.

- Non può vivere da solo: ha bisogno di un ospitante
- Inserisce una **firma** (una specie di ID) per capire se un file è già stato infettato

### Caso studio: Brain virus

È uno dei primi virus per PC, scoperto nel 1986, progettato per infettare **sistemi MS-DOS**. Era un **virus di boot sector**, quindi infettava la parte del disco da cui il PC avviava il sistema.

- Cambia il nome del volume del disco in **BRAIN**, così te ne accorgi solo se lo leggi.
  - Il virus **si auto-propaga** su altri dischetti/dati che legge.
- 

#### 1. Si carica in memoria alta

- Si installa nella parte "alta" della RAM (sopra i 640 KB).
- Usa una **chiamata di sistema** per **nascondersi**, dicendo al sistema operativo che **la RAM finisce prima** (così non viene sovrascritto).

#### 2. Aggancia l'interrupt Ox19

- In MS-DOS, l'interrupt **Ox19** gestisce le **lettura da disco**.
  - Brain **sostituisce l'indirizzo originale dell'interrupt** con il proprio.
  - Risultato: **ogni lettura da disco passa prima per il virus**, anche quella del **boot sector**!
- 

Quando intercetta una lettura da disco:

Se i byte 5 e 6 contengono **Ox1234**:

- È la **firma del virus** → il settore è già infetto.
- Non fa nulla.

Se non c'è la firma:

- Infetta il disco, scrivendo il proprio codice in **6 settori a caso**.
  - Firma i settori con **Ox1234** per ricordarsi che sono "suoi".
- 

Nelle versioni più aggressive:

- **Marca i settori infetti come "bad sectors".**
    - Così il sistema operativo li **considera danneggiati e non li usa più**.
  - Itera il processo: cerca continuamente **settori non infetti** e li infetta.
  - Col tempo, lo **spazio disponibile sul disco cala** sempre di più.
- 

## Chi lo ha creato e perché

È stato scritto da **due fratelli pakistani, Amjad e Basit Farooq Alvi**, non con scopi malevoli, ma per **proteggere il proprio software medico** dalla copia illegale.

Il virus includeva addirittura **il numero di telefono del loro studio**, come a dire: "Contattaci se trovi questo virus!".

Quindi era un **prototipo rudimentale di DRM** (Digital Rights Management) fatto in casa. Ma si diffuse rapidamente in modo incontrollato, diventando la **prima "epidemia informatica" su scala globale**.

---

## Macrovirus

Un tipo di virus che sfrutta **le macro** (script automatici) in programmi come **Microsoft Office**.

- Esempio classico: virus scritti in VBA (Visual Basic for Applications) dentro un documento Word
- Negli anni '90 era comunissimo, oggi molto meno grazie a restrizioni sulle macro

Esempio: apri un file Word → parte una macro che installa un trojan.

## Antivirus

Gli antivirus si sono evoluti nel tempo in **4 generazioni principali**:

### 1<sup>a</sup> generazione – Basata su firma

- Scansionano i file cercando **firme note di virus** nel loro database
- Controllano **la lunghezza** dei file
- I virus provavano a **camuffarsi con tecniche di compressione**

### 2<sup>a</sup> generazione – Basata su euristiche

- Ricerca **pattern sospetti** (frammenti di codice tipici di virus)
- Controlli di **integrità** (hash, checksum)

### 3<sup>a</sup> generazione – Basata sul comportamento

- Monitorano **azioni sospette** in tempo reale:
  - apertura massiva di shell
  - grandi cancellazioni di file
  - scrittura in settori critici

### 4<sup>a</sup> generazione – Integrata nella sicurezza del sistema

- Tutto ciò che fanno le generazioni precedenti +
- **Modifica i permessi di accesso** per bloccare la diffusione
- Es: se rileva un virus, toglie i **permessi di scrittura** a tutti per evitare l'infezione

## SSL

**Secure Sockets Layer** è un protocollo di sicurezza fra livello trasporto e livello applicazione. Ma sia SSL che TLS possono essere implementati ad ambedue i livelli.

- **HTTPS = HTTP + SSL/TLS**  
È la versione **sicura** di HTTP. Grazie a SSL/TLS, tutte le comunicazioni tra il browser e il server sono **cifrate, autenticate e integre**.
- **Altri protocolli che usano SSL/TLS:**
  - **S/MIME** (per email sicure)
  - **3D-Secure** (per pagamenti sicuri, tipo Verified by Visa)

### HSTS e gli attacchi SSL Stripping

**HSTS (HTTP Strict Transport Security)** è un *header speciale* che dice al browser:

"Hey, quando visiti questo sito, usalo solo in HTTPS, mai in HTTP!"

### Attacco SSL Stripping

1. **Versione 1** (prevenuta da HSTS):
  - Attaccante (Man-in-the-middle) intercetta la richiesta e fa vedere il sito in **http**, non cifrato.
  - Se il browser ha visto quell'host prima senza HSTS, ci casca.
  - Ma se il sito ha inviato **HSTS in precedenza**, il browser si rifiuta di accettare la versione http → sicuro.

## 2. Versione 2 (non prevenuta da HSTS):

- Attaccante ti manda un sito **simile ma falso** (es: `g00gle.com` o con un certificato truccato)
- Può essere in `http` oppure anche in `https` con un certificato farlocco.
- Questo richiede **phishing + social engineering** → HSTS non può aiutare.

### SSL/TLS: com'è fatto?

SSL/TLS si posiziona tra il **livello applicazione (es. HTTP)** e il **livello trasporto (TCP)**.

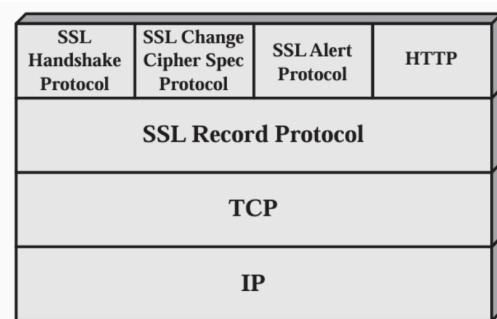
Serve a **proteggere i dati** durante la comunicazione.

### SSL e TLS: sono la stessa cosa?

- **SSL** è la vecchia versione.
- **TLS** è la nuova (ma compatibile).
- Es: **TLS 1.0 = SSL 3.1**

Quindi quando un server ti dice "uso SSL 3.1", in realtà intende "TLS 1.0"

### I protocolli interni di SSL/TLS



#### 1. SSL Handshake Protocol

Serve per:

- Autenticare il server (e a volte il client)
- Scegliere l'algoritmo di cifratura
- Generare la chiave di sessione (tipo Diffie-Hellman)

#### 2. SSL Change Cipher Spec Protocol

È il segnale che dice "ok, da ora cifriamo tutto".

Deve essere **cifrato**, altrimenti un attaccante potrebbe forzare un **downgrade** (es: usare una versione più vecchia e vulnerabile di SSL).

#### 3. SSL Alert Protocol

Serve per inviare **messaggi d'errore o di stato**.

Contiene:

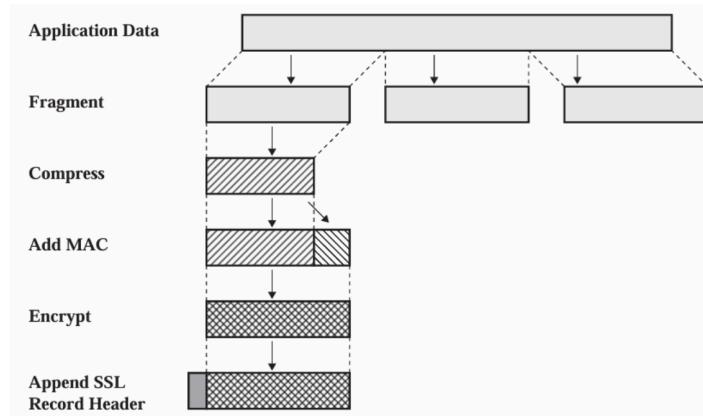
- **Livello:** warning o fatal
- **Codice:** tipo di errore

Esempi:

Livello	Codice	Significato
fatal	<code>bad_record_mac</code>	MAC non valido (messaggio modificato)
fatal	<code>decompression_failure</code>	Fallita decompressione
warning	<code>cert_expired</code>	Certificato scaduto

Se è **fatal**, la connessione viene **chiusa immediatamente**.

### SSL Record Protocol



Questo è il "trasportatore" vero e proprio: gestisce i dati **una volta stabilita la sessione cifrata**.

#### Come funziona?

1. I dati vengono:

- **Frammentati**
- (Facoltativamente) **compressi**
- Viene calcolato un *MAC* → serve per verificare l'integrità
- Viene **cifrato** il tutto
- Si aggiunge l'**SSL Record Header**

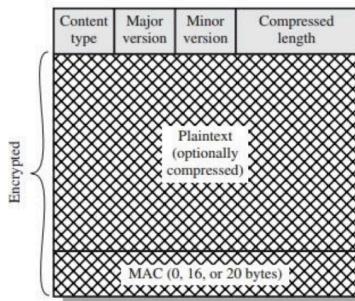
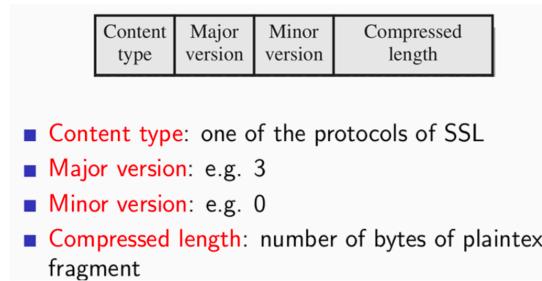


Figure 16.4 SSL Record Format

Il **MAC** è obbligatorio, proprio per garantire integrità → Non può essere vuoto (0 byte).

#### Come viene calcolato il *MAC*?

Formula:

```

hash(
    MAC_write_secret || pad2 ||
    hash(
        MAC_write_secret || pad1 || seq_num || SSLCompressed.type ||
        SSLCompressed.length || SSLCompressed.fragment
    )
)

```

Traduco:

- È un **doppio hash** (interno + esterno)
- Serve a evitare attacchi su hash troppo deboli
- Dentro ci stanno:
  - La chiave segreta per il *MAC* (`MAC_write_secret`)
  - Il tipo di messaggio (es: handshake, data, alert...)
  - La lunghezza
  - Il messaggio vero e proprio
  - Un numero di sequenza per proteggere contro replay

## SSL Handshake Protocol

Lo scopo del protocollo è **stabilire in modo sicuro una chiave di sessione condivisa**, fornendo:

- **Segretezza**: il contenuto è cifrato. Anche se:
  - Le **prime fasi sono in chiaro** (salvo la parte del `pre_master_secret`, se RSA).
  - **Solo l'ultima parte è cifrata**, dopo aver derivato le chiavi di sessione.
- **Autenticazione**: le parti sanno con chi stanno parlando.
- **Integrità**: nessuno può modificare il messaggio senza essere scoperto.

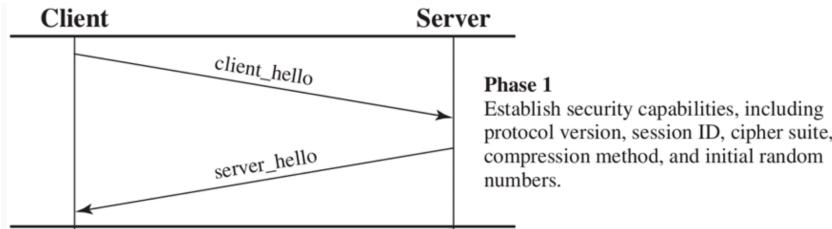
Tutto questo avviene **su una rete potenzialmente insicura** (es: Internet), quindi è un vero capolavoro di ingegneria.

Message Type	Parameters
<code>hello_request</code>	null
<code>client_hello</code>	version, random, session id, cipher suite, compression method
<code>server_hello</code>	version, random, session id, cipher suite, compression method
<code>certificate</code>	chain of X.509v3 certificates
<code>server_key_exchange</code>	parameters, signature
<code>certificate_request</code>	type, authorities
<code>server_done</code>	null
<code>certificate_verify</code>	signature
<code>client_key_exchange</code>	parameters, signature
<code>finished</code>	hash value

---

### Le 4 Fasi dell'Handshake SSL

#### Fase 1 – Negoziazione dei parametri di sicurezza



#### Client → Server: ClientHello

Contiene:

- La **versione SSL** più alta supportata.
- Un **nonce random** generato dal client.
- Un **Session ID** (vuoto o meno, serve per il resume).
- Una lista di **Cipher Suites** supportate.
- Lista dei metodi di compressione.

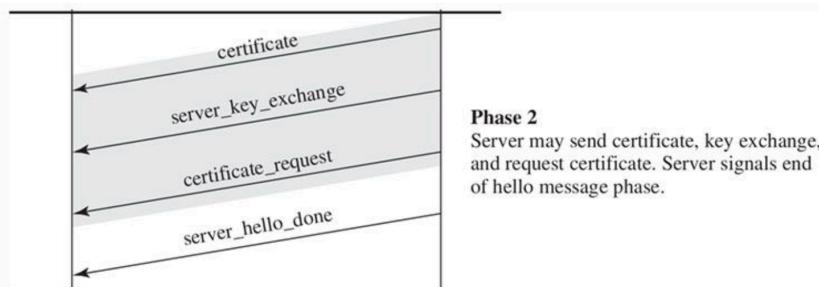
#### Server → Client: ServerHello

Risponde con:

- Versione più alta **comune** tra client e server.
- Il proprio **nonce random**.
- Il Session ID da usare.
- La Cipher Suite **scelta** (tra quelle proposte).
- Il metodo di compressione scelto.

I nonce random servono per garantire freshness ed evitare attacchi di replay.

#### Fase 2 – Autenticazione del server e scambio delle chiavi

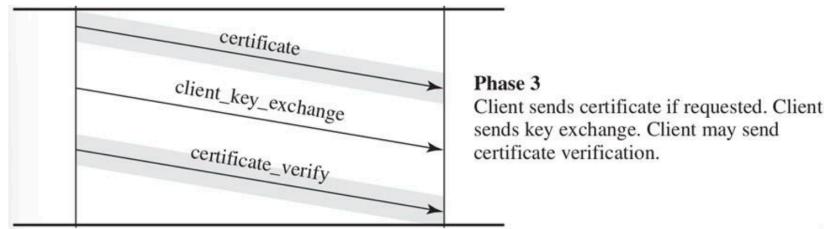


#### Server → Client

- **Certificate**: il certificato X.509 del server.
- **ServerKeyExchange (opzionale)**: solo per certi algoritmi (es: Ephemeral Diffie-Hellman).
- **CertificateRequest (opzionale)**: se il server vuole autenticare anche il client.
- **ServerHelloDone**: chiusura della fase.

**Autenticazione del server**: il client verifica che il certificato sia valido e appartenga davvero al server.

#### Fase 3 – Autenticazione del client e chiave di sessione

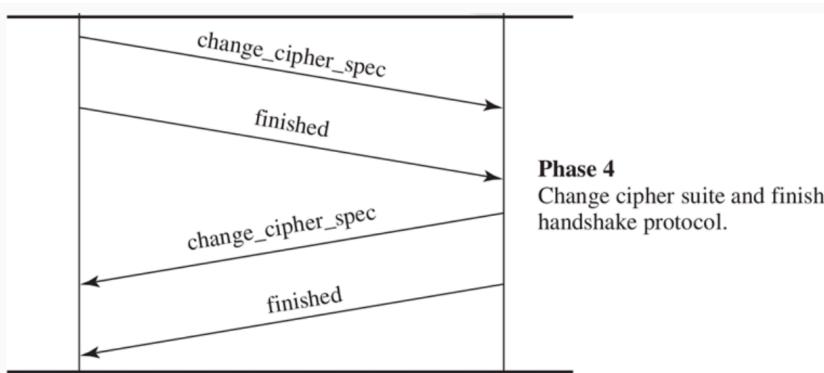


#### Client → Server

- **Certificate (opzionale)**: se richiesto dal server.
- **ClientKeyExchange**: invia il **Pre-Master Secret** (dipende dal Key Exchange usato).
  - Se si usa **RSA**: il client cifra il PMS con la **chiave pubblica del server**.
  - Se si usa **Diffie-Hellman Anonimo**: versione tradizionale, MITM.
  - Se si usa **Ephemeral Diffie-Hellman**: parametri pubblici autenticati dalla firma digitale.
  - Se si usa **Fixed Diffie-Hellman**: i parametri pubblici DH sono fixed, derivati dai certificati del server e del client.
  - Fortezza: adesso deprecata
- **CertificateVerify (opzionale)**: una firma digitale che dimostra la proprietà della chiave privata.

Il **Pre-Master Secret (PMS)** è il cuore della sicurezza. Da questo, usando i random visti prima, si ricava il **Master Secret**, la chiave di base per tutta la sessione.

#### Fase 4 – Attivazione della cifratura



#### Client → Server: `ChangeCipherSpec`

- Il client dice: "da ora in poi cifro i messaggi".

#### Client → Server: `Finished`

- Primo messaggio **cifrato**, che dimostra che ha calcolato correttamente la chiave.

#### Server → Client: `ChangeCipherSpec` e `Finished`

- Anche il server inizia a cifrare e conferma che è tutto a posto.

#### Derivazione del Master Secret (in SSL)

Avviene così:

```
master_secret = MD5(pre_master_secret || SHA('A' || pre_master_secret ||
                                              ClientHello.random || ServerHello.random)) ||
                  MD5(pre_master_secret || SHA('BB' || pre_master_secret ||
                                              ClientHello.random || ServerHello.random)) ||
                  MD5(pre_master_secret || SHA('CCC' || pre_master_secret ||
                                              ClientHello.random || ServerHello.random))
```

Risultato? Un segreto da **48 byte** condiviso tra client e server, **mai scambiato direttamente sulla rete**.

- Il `master_secret` **non cifra direttamente i dati**
- Viene usato **solo per derivare il `key_block`**, da cui si estraggono le chiavi vere per cifratura e integrità

### Derivazione delle chiavi di sessione

Dal Master Secret si ricava il **key\_block**, che contiene:

- `client_write_MAC_secret` → chiave per calcolare il *MAC* dei messaggi **inviai dal client**
- `server_write_MAC_secret` → chiave per calcolare il *MAC* dei messaggi **inviai dal server**
- `client_write_key` → chiave di **cifratura** usata dal client
- `server_write_key` → chiave di **cifratura** usata dal server
- `initialization_vectors` → usato per inizializzare la codifica a blocchi
  - L'**IV** è un valore casuale (o pseudo-casuale) che viene usato **insieme alla chiave, solo per il primo blocco**. A seconda della modalità di cifratura, l'IV serve per **"mescolare"** i **dati in modo sicuro** e impedire che messaggi uguali producano blocchi cifrati uguali.

L'IV è usato per inizializzare la codifica a blocchi, cioè serve a rendere la cifratura non deterministica e quindi più sicura, anche quando si inviano messaggi ripetitivi.

```
key_block = MD5(master_secret || SHA('A' || master_secret ||  
    ServerHello.random || ClientHello.random)) ||  
    MD5(master_secret || SHA('BB' || master_secret ||  
    ServerHello.random || ClientHello.random)) ||  
    MD5(master_secret || SHA('CCC' || master_secret ||  
    ServerHello.random || ClientHello.random)) || ...
```

E anche qui si usano iterazioni tipo la formula del `master_secret`, ma con i random **invertiti** (prima `ServerHello.random`, poi `ClientHello.random`).

L'ordine viene invertito tra `master_secret` e `key_block` per motivi di sicurezza e buona progettazione: aiuta a separare chiaramente le fasi, evita collisioni tra chiavi, e migliora la robustezza contro implementazioni errate o attacchi.

### Ripresa di sessione (Session Resumption)

Per velocizzare il riutilizzo di una connessione recente, si possono **saltare le fasi 2 e 3** se la sessione "is resumable" e se:

- Il **Session ID** è ancora valido.
- Il server ha ancora memorizzato i dati (o usa session ticket).

Vengono quindi riusati:

- Master secret
- Cipher suite
- Compression method
- Peer certificate (certificato X.509 del server, ed eventualmente del client)

## TLS

**TLS (Transport Layer Security)** nasce come l'evoluzione standardizzata di **SSL 3.0**.

Non è un protocollo completamente nuovo, ma piuttosto un raffinamento di SSL, con miglioramenti su robustezza crittografica e flessibilità.

TLS 1.0 = SSL 3.1

(giusto per dare l'idea della continuità)

### Le principali differenze tra SSL e TLS

Aspetto	SSL 3.0	TLS 1.0
MAC	MAC personalizzato	HMAC (standard)
Derivazione delle chiavi	Meccanismo artigianale con MD5 e SHA	PRF (Pseudo Random Function)
Cipher Suite	Set limitato	Più ricco e negoziabile
Alert	Meno dettagliati	Più completi (nuovi codici di errore)
Estensioni	Non previste	Aggiunte da TLS 1.2 in poi

### L'HMAC in TLS

Una delle **prime novità chiave** è l'introduzione dell'*HMAC* (Hash-based Message Authentication Code), che rimpiazza il *MAC* di SSL.

#### Formula dell'*HMAC*:

$$HMAC_K(M) = H[(K^+ \oplus opad) \parallel H[(K^+ \oplus ipad) \parallel M]]$$

Dove:

- $H$  è una funzione hash (MD5 o SHA-1 in TLS 1.0).
- $K^+$  è la chiave segreta estesa a 512 bit (con padding a sinistra).
- $M$  è il messaggio da autenticare.
- $\oplus$  è lo XOR.
- $ipad$  è  $0x36$  ripetuto 64 volte (equivalente di pad1 in SSL)
- $opad$  è  $0x5C$  ripetuto 64 volte (equivalente di pad2 in SSL)

#### Differenza dal *MAC* in SSL:

Nel *MAC* di SSL c'è annidamento tra due hash, ma **manca l'uso dello XOR con padding**, che è ciò che rende l'*HMAC* più robusto contro attacchi di tipo collisione.

Lo XOR è un'operazione che, se fatta con chiavi random **fresche**, è **molto sicura**.

### La Pseudo Random Function (PRF)

Un altro elemento **cruciale** introdotto da TLS è la **PRF**, che gestisce:

- La generazione del **master secret**.
- La derivazione dei **key block** (MAC secret, encryption key, ecc.).

#### Formula base:

$$\text{PRF}(\text{secret}, \text{label}, \text{seed}) = P\text{-MD5}(S1, \text{label} \parallel \text{seed}) \text{ XOR } P\text{-SHA-1}(S2, \text{label} \parallel \text{seed})$$

- $\text{secret}$ : è il pre-master secret (o master secret, a seconda del contesto).
- $\text{label}$ : stringa costante (es: "master secret", "key expansion").
- $\text{seed}$ : in genere è la concatenazione di  $\text{ClientHello.random} \parallel \text{ServerHello.random}$ .
- $S1$  e  $S2$ : sono la **prima e seconda metà** del segreto.

#### Cos'è $P\text{-hash}$ ?

La  $P\text{-HASH}$  (**non è la PRF**) è una **funzione ausiliaria** definita in TLS come:

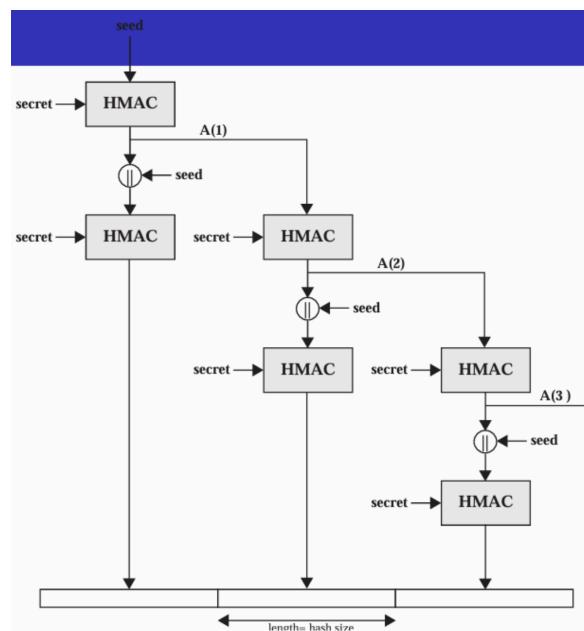
$$\begin{aligned} P\text{-HASH}(\text{secret}, \text{seed}) &= \text{HMAC\_hash}(\text{secret}, A(1) + \text{seed}) + \\ &\quad \text{HMAC\_hash}(\text{secret}, A(2) + \text{seed}) + \end{aligned}$$

```
HMAC_hash(secret, A(3) + seed) + ...
```

dove:

- **A(x)** è una catena di HMAC calcolati uno sull'altro, ciascuno che prende come input il risultato del precedente.
- $A(0) = \text{seed}$
- $A(i) = \text{HMAC\_hash}(\text{secret}, A(i-1))$

Si tratta quindi di un **meccanismo di espansione pseudo-casuale** basato su *HMAC* (con hash = MD5 o SHA-1 in TLS 1.0/1.1, SHA-256 o altri in TLS 1.2).



Questo approccio rende la derivazione più sicura e resistente ad attacchi su hash o pattern prevedibili.

In TLS è **obbligatorio usare la PRF**. L'uso di **P\_HASH** è implicito, perché è parte integrante dell'implementazione della PRF stessa.

#### Come funziona tutto questo nella pratica?

Vediamolo applicato ai due casi principali.

##### Generazione del Master Secret

```
master_secret = PRF(pre_master_secret, "master secret", ClientHello.random || ServerHello.random)
```

- 48 byte totali
- Il `pre_master_secret` viene dalla fase di key exchange
- I nonce random servono come sale per la PRF

##### Generazione del Key Block

```
key_block = PRF(master_secret, "key expansion", ServerHello.random || ClientHello.random)
```

- Nota l'inversione dei random rispetto a prima!
- Dal `key_block` ricaveremo:
  - `client_write_MAC_secret`
  - `server_write_MAC_secret`

- client\_write\_key
- server\_write\_key
- (e volendo anche IV se serve)

### Evoluzioni successive

Con **TLS 1.2**, sono state aggiunte:

- Estensioni nei messaggi Hello (come **HSTS**)
- Possibilità di **negoziare l'algoritmo di hash**

Inoltre la PRF non è più quella fissa, ma può essere costruita con **qualsiasi algoritmo hash sicuro** (tipo SHA-256 o SHA-384).

### Aggiunta: HSTS e vulnerabilità

HSTS: è un'estensione di **TLS**, che impone l'uso forzato di HTTPS.

**TLS 1.0 e 1.1** sono vulnerabili a **SSL Stripping**, perché un attaccante potrebbe intercettare la prima richiesta HTTP e impedire il redirect a HTTPS. Con **HSTS**, invece, il browser sa già che il sito va aperto in HTTPS e non si fida del primo messaggio ricevuto.

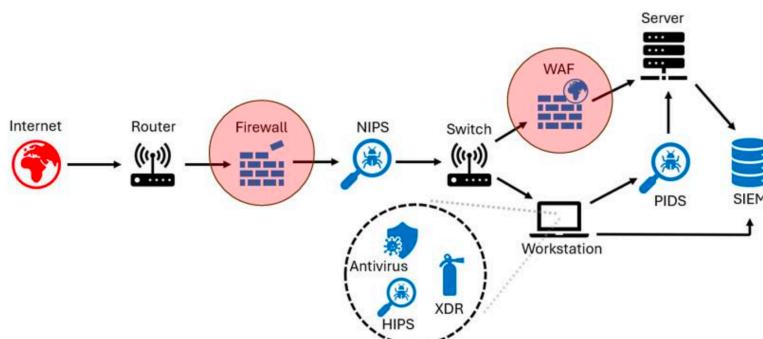
## Difesa della rete

### Cos'è un'intrusione?

Un'intrusione avviene quando **un utente riesce ad ottenere privilegi che non gli spettano**, ad esempio accedendo come amministratore, o leggendo/modificando dati a cui non dovrebbe accedere.

### Difesa: servono più livelli

Poiché esistono **tanti modi per attaccare**, dobbiamo costruire **più linee di difesa**, come una **fortezza a strati**. Ogni strato protegge il successivo.



### Definizioni chiave per capire la difesa

#### • Vulnerabilità:

Una **vulnerabilità** è una **debolezza** in un sistema informatico (hardware, software, rete, configurazione o processo) che può essere **sfruttata da una minaccia** per violare la **sicurezza** del sistema.

È un punto debole che, se scoperto e sfruttato da un attaccante, può compromettere **riservatezza, integrità o disponibilità** delle informazioni o dei servizi.

Una vulnerabilità è **concreta**, ovvero:

- **Reale**: presente in un software, dispositivo o configurazione realmente in uso.
- **Attaccabile**: qualcuno può davvero **scrivere un exploit** per sfruttarla.
- **Verificata**: è stata **dimostrata o osservata** in un attacco, o test di sicurezza.

Le vulnerabilità note sono raccolte nel **CVE (Common Vulnerabilities and Exposures)**, un archivio mantenuto dal **MITRE**, che assegna un ID (es: CVE-2023-12345).

- **Debolezza (CWE):**

È un **problema di progettazione generale, teorico o astratto, non legato a un software specifico.**

Esempio: "uso di funzioni di memoria non sicure in C" è una debolezza.

Le debolezze sono raccolte nel **CWE (Common Weakness Enumeration)**, sempre gestito da MITRE.

- **Exploit:**

È **qualunque strumento, codice, stringa o programma che sfrutta una vulnerabilità per ottenere un vantaggio illecito.**

Es: un programma che manda un input malevolo per far eseguire codice arbitrario.

Debolezza (CWE) → può portare a una → Vulnerabilità (CVE) → che può essere sfruttata tramite un → Exploit

### E per difenderci?

- Dobbiamo **monitorare le vulnerabilità CVE**
- Evitare **debolezze CWE** nel codice (scrivere codice sicuro!)
- Applicare **patch e aggiornamenti**
- Usare sistemi di **Intrusion Detection e Prevention**
- Avere una **difesa a strati**: firewall, autenticazione forte, logging, crittografia, controllo accessi, ecc.

## Firewall

È un sistema (hardware o software) che controlla il traffico tra due reti con diversi livelli di sicurezza (es. LAN ↔ Internet), applicando delle regole (*policy*) per decidere cosa far passare e cosa bloccare.

### Requisiti fondamentali per funzionare:

1. **Tutto il traffico** tra le due reti deve passare attraverso il firewall.
2. Il traffico deve essere regolato da **regole ben definite**.
  - a. Le regole possono essere specificate per gruppi. Ad esempio, "ammetti tutti i pacchetti TCP e UDP in entrata e uscita" copre tutti i tipi di pacchetti.

### Funzioni principali di un firewall:

- **Protezione dei servizi**: esporre porte solo a reti fidate.
- **Monitoraggio e filtraggio**: bloccare pacchetti sospetti (es. bind shell).
- **Filtraggio contenuti**: es. allegati e-mail pericolosi.

### Funzioni aggiuntive:

- VPN tramite IPsec
- DHCP (client/server)
- NAT (traduzione indirizzi IP)

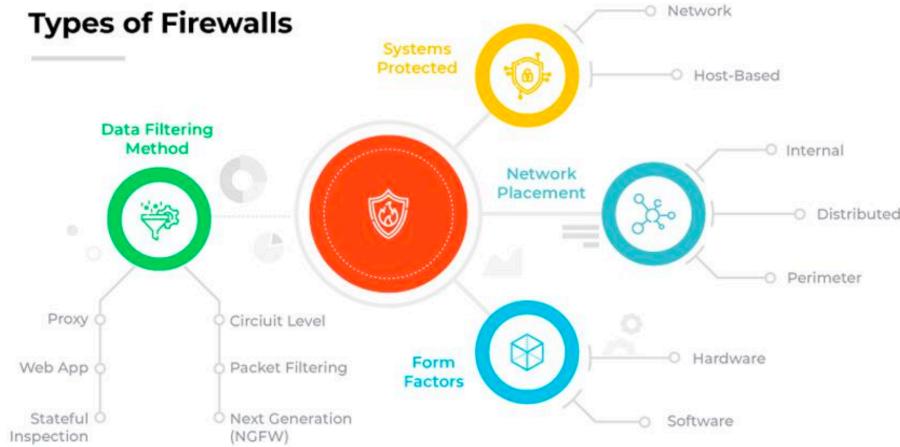
### Attenzione! Anche i firewall hanno vulnerabilità.

Esempi:

- **CVE-2024-3400**: code injection da remoto, anche senza autenticazione.
- **CVE-2022-30525**: modifica di file per eseguire comandi da remoto.

### Tipologie di Firewall

## Types of Firewalls



### In base a cosa proteggono (Systems Protected):

- **Network firewall:** tra reti diverse (es. LAN ↔ Internet). Serve a proteggere una o più reti e a mantenerne l'integrità.
- **Host-based firewall:** su singolo dispositivo. Anche se la rete viene violata, l'Host-Based Firewall può fornire una seconda linea di difesa per il dispositivo.

### In base alla posizione nella rete (Placement):

- **Internal:** tra dispositivi nella stessa rete
- **Distributed:** firewall distribuiti su più macchine
- **Perimeter:** al confine tra LAN e Internet

### In base alla forma (Form Factor):

- **Hardware firewall:** dispositivo fisico (e.g., connettendolo al router e permettendo ai dispositivi di accedere ad Internet solo attraverso il firewall).
- **Software firewall:** programma installato su macchina. Utile per scenari in cui non è possibile (o è difficile) usare firewall fisici (e.g. cloud, container). Essendo dei software a tutti gli effetti, è possibile estenderne le funzionalità nel tempo.

### Metodi di filtraggio dei dati (Data Filtering Method)

#### 1. Packet Filtering Firewall (PFF):

- Opera al livello 3 (rete)
- Si basa su regole su IP, porte, protocolli
- **Limi:**
  - **Stateless:** non tiene traccia delle connessioni, analizza solo ogni pacchetto **singolarmente**.
  - **Regole rigide:** fa passare solo i pacchetti che corrispondono alle regole "allow", scartando gli altri.
  - **Facilmente ingannabile:** le informazioni nei pacchetti (come IP o porte) possono essere **modificate** da un attaccante per **bypassare le regole**.
  - **Vulnerabile ad attacchi di routing:** un attaccante può far passare il traffico attraverso un host già autorizzato, **ingannando il firewall**.

#### 2. Stateful Inspection Firewall (SIF):

- Opera ai livelli 3-4 (rete e trasporto)
- Tiene traccia delle connessioni
- Può fare **Deep Packet Inspection**, con due tecniche:
  - **Protocol Anomaly:** Di default non viene fatto passare nessun pacchetto ("default deny"). Solo i pacchetti che corrispondono ad un determinato set di regole possono passare.

- **Pattern/Signature Matching:** Di default vengono fatti passare tutti i pacchetti ("default permit"). I pacchetti che corrispondono ad un determinato set di regole vengono bloccati.

### 3. Circuit-Level Gateway:

- Livello 5 (sessione)
- Verifica della genuinità degli handshake TCP e degli attori coinvolti;
- Creazione di un circuito virtuale per la durata della sessione;
- Filtraggio dei pacchetti sugli indirizzi e le porte specificate, a seconda della loro validità nel contesto della sessione.

#### Vantaggi:

- Nasconde gli IP interni: il traffico in uscita sembra provenire solo dal **gateway**, proteggendo l'anonimato della rete.
- Facile da configurare e **non rallenta la rete** (anzi, può persino migliorarla!).

#### Svantaggi:

- Filtra solo a livello di sessione: se una sessione è stata autorizzata, **qualsiasi pacchetto** (anche pericoloso) può passare.
- Rischio malware: **contenuti malevoli** possono viaggiare indisturbati nel canale.

### 4. Proxy Firewall:

- Lavora al **livello 7 (applicazione)**.
- Agisce come un **Man-In-The-Middle** (MITM) tra client interni e server esterni.
- Usa un **proprio indirizzo IP, nascondendo l'identità** dei dispositivi interni alla rete.
- Può **analizzare a fondo il contenuto dei pacchetti** (deep packet inspection) prima di inoltrarli.

### 5. Web Application Firewall (WAF):

- Funziona al **livello 7 (applicazione)**
- Fa da **reverse proxy** tra i **client esterni** e i **server interni**.
- **Protegge l'identità dei server interni**: i client vedono solo il WAF, non i server reali.
- Effettua **Deep Packet Inspection** per individuare **payload noti** usati negli attacchi.
  - '`or 1=1 --+` → **SQL Injection**
  - '`;cat /etc/passwd` → **OS Command Injection**
- Dietro al WAF possono esserci **più server**, ma dall'esterno sembrano uno solo.

### Next-Generation Firewall (NGFW):

- **Combina più funzionalità di firewall** tradizionali in un'unica soluzione.
- **Non esiste una definizione precisa**: ogni NGFW può includere strumenti diversi.
  - Packet filtering
  - TLS/SSL inspection
  - Intrusion Prevention System (IPS)
  - Protezione da malware/threats
  - Deep Packet Inspection
  - Analisi del traffico applicativo
- Gli NGFW sono **firewall "tuttofare"**: mirano a **unire sicurezza di rete tradizionale e avanzata**.
- Sono **modulari e flessibili**, spesso usati in ambienti complessi e moderni.

## Netfilter

**Netfilter** è un **framework del kernel Linux** che permette di **intercettare, analizzare e modificare** i pacchetti di rete, direttamente all'interno dello **stack di rete**.

È ciò su cui si basano strumenti famosi come **iptables**, **nftables**, **ufw**, ecc.

In pratica: quando un pacchetto passa attraverso il kernel, Netfilter offre dei **punti di aggancio** (detti *hook*) dove è possibile "attaccare" delle funzioni che decidono cosa fare con quel pacchetto (lasciarlo passare, bloccarlo, modificarlo, reindirizzarlo...).

## I 5 hook principali di Netfilter

Ogni **hook** corrisponde a una fase specifica del viaggio del pacchetto nello stack di rete. Eccoli:

- **NF\_IP\_PRE\_ROUTING**: Aggancia i pacchetti in entrata, subito dopo il loro ingresso nel network stack.
- **NF\_IP\_LOCAL\_IN**: Aggancia i pacchetti in entrata, se questi sono destinati al sistema locale.
- **NF\_IP\_FORWARD**: Aggancia i pacchetti in entrata che devono essere inoltrati ad un altro host.
- **NF\_IP\_LOCAL\_OUT**: Aggancia i pacchetti in uscita che vengono creati dal sistema locale, subito dopo il loro ingresso nel network stack.
- **NF\_IP\_POST\_ROUTING**: Aggancia i pacchetti in uscita (outgoing + forward) subito prima che questi vengano spediti.

## Iptables

**iptables** è uno **strumento a riga di comando** su Linux che permette di **configurare il firewall** del sistema.

Funziona **interfacciandosi direttamente con Netfilter**, cioè il framework interno al kernel che intercetta e gestisce i pacchetti di rete.

È un **firewall software** di tipo:

- **Host-based** (funziona sul singolo dispositivo)
- **Packet filter** (decide cosa fare con ogni pacchetto)
- Con **stateful inspection** (sa se un pacchetto appartiene o no a una connessione esistente)

### Come funziona iptables?

iptables organizza le sue regole in **tabelle**, ognuna delle quali ha uno **scopo specifico**.

### Le 5 tabelle principali:

Tabella	Scopo principale	Quando la usi
filter	<b>Filtraggio puro</b> (la più usata!)	Vuoi decidere se accettare o bloccare un pacchetto
nat	<b>NAT (Network Address Translation)</b>	Vuoi cambiare l'IP sorgente o destinazione
mangle	<b>Modifica degli header dei pacchetti</b>	Vuoi alterare TTL, Type of Service, ecc.
raw	<b>Esclusioni dal connection tracking</b>	Vuoi dire a conntrack "non tracciare questo pacchetto"
security	<b>Marcatura dei pacchetti con contesti di sicurezza</b>	Usi SELinux, AppArmor, ecc.

### Dettaglio rapido delle ultime tre:

- **Mangle**: per esempio puoi abbassare il TTL per evitare loop infiniti o impedire certi attacchi DoS.
- **Raw**: utile quando vuoi saltare completamente lo stato della connessione (es. per motivi di performance o debugging).
- **Security**:
  - **SECMARK**: assegna un'etichetta di sicurezza al singolo pacchetto
  - **CONNSECMARK**: etichetta l'intera connessione a cui appartiene il pacchetto

### Catene e hook

Dentro ogni tabella ci sono **catene** (*chains*), che rappresentano **fasi precise del percorso di un pacchetto**, agganciate agli **hook** di Netfilter che abbiamo visto prima.

Le **catene predefinite sono**:

Catena	Quando viene eseguita	Hook corrispondente
<b>PREROUTING</b>	Appena arriva un pacchetto sulla rete	NF_IP_PRE_ROUTING
<b>INPUT</b>	Quando il pacchetto è per la macchina locale	NF_IP_LOCAL_IN
<b>FORWARD</b>	Quando il pacchetto deve essere inoltrato	NF_IP_FORWARD
<b>OUTPUT</b>	Quando la macchina genera un pacchetto	NF_IP_LOCAL_OUT
<b>POSTROUTING</b>	Prima che il pacchetto esca fisicamente dalla rete	NF_IP_POST_ROUTING

Oltre a queste puoi creare **catene personalizzate**, ma queste non sono agganciate direttamente a un hook. Devi "chiamarle" tu da una catena predefinita tramite una regola.

### Conntrack e Stateful Inspection

iptables **non lavora da solo**, ma usa un modulo chiamato **conntrack** per fare il tracking delle connessioni.

Questo modulo tiene traccia dello **stato** di ogni pacchetto. Gli stati principali sono:

Stato	Significato
NEW	Il pacchetto è relativo ad una nuova connessione
ESTABLISHED	Il pacchetto è relativo ad una connessione già esistente
RELATED	Il pacchetto è correlato ad una connessione già esistente, ma non ne fa parte (es. la porta dati FTP)
INVALID	Il pacchetto non è relativo ad una nuova connessione o ad una connessione già esistente

Grazie a questo, iptables può dire cose tipo:

```
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
```

Che vuol dire: *accetta i pacchetti se fanno parte di una connessione già esistente o legittimamente collegata.*

### Problemi di iptables

Con gli anni, **iptables ha mostrato dei limiti**, ecco i più importanti:

#### 1. Tutte le tabelle e catene create di default

Anche se usi solo la tabella **filter**, tutte le tabelle e catene vengono comunque **caricate**.

Questo può portare a **spreco di risorse e performance peggiori**, specie in sistemi con molto traffico.

#### 2. IPv6 gestito separatamente

- Per IPv4 usi **iptables**
- Per IPv6 devi usare **ip6tables**

Due comandi diversi, **nessuna sincronizzazione** automatica: se fai una regola su uno, **non vale per l'altro** → devi replicare tutto a mano.

#### 3. Supporto a nuovi protocolli richiede update del kernel

Se esce un nuovo protocollo (es. QUIC o roba personalizzata), iptables non lo può gestire **senza aggiornare il kernel**.

Questo può causare:

- **Incompatibilità**
- **Downtime**
- Problemi con tool e servizi già in uso

E infatti... è nato **nftables**

## Nftables

**nftables** è il successore di **iptables**, introdotto nel **2014** per **risolvere tutti i suoi problemi storici**.

Lavora **sempre con Netfilter**, quindi sotto il cofano funziona alla stessa maniera, **ma è molto più flessibile, modulare e potente**.

In pratica:

- Unifica IPv4 e IPv6
- Scrivi regole più compatte e leggibili
- Nessuna struttura è creata automaticamente → **sei tu a decidere tutto**
- Supporta un **sistema di priorità**, che ti dà controllo completo su cosa succede e quando
- Hook Ingress: aggancia i pacchetti dopo che questi vengono rilasciati dal NIC (Network Interface Controller), ovvero, ancora prima del prerouting.

### Come funziona nftables?

A differenza di iptables (dove ogni cosa era già preconfigurata), in nftables **non esiste nulla finché non lo crei tu**.

#### Tre tipi di tabelle principali:

Tabella	Scopo
filter	Filtraggio puro dei pacchetti
nat	NAT e modifica di indirizzi IP/porta
route	Come mangle in iptables, per modificare headers ecc.

#### Famiglie di protocolli

Quando crei una tabella, **la associ a una famiglia**, cioè il tipo di traffico che vuoi filtrare.

Famiglia	Descrizione
ip	Solo IPv4
ip6	Solo IPv6
inet	<b>IPv4 + IPv6 insieme (super comodo!)</b>
arp	Per pacchetti ARP
bridge	Per reti bridge (es. container)
netdev	Aggancia i pacchetti su una specifica interfaccia e possiamo usare l'hook ingress, ad esempio per droppare DDoS

inet permette di scrivere una volta sola regole valide sia per IPv4 che IPv6!

#### Le catene

In nftables esistono **due tipi di catene**, più flessibili rispetto a iptables:

Tipo di catena	Cosa fa
Base chain	È direttamente collegata a un <b>hook</b> di Netfilter (es: input, output, forward, ingress, ecc.). Serve per <b>agganciare la tua catena al punto giusto</b> del network stack.
Regular chain	È come una catena "user-defined". La esegui solo se ci fai un <b>jump</b> da un'altra catena.

Grande vantaggio: puoi avere più base chain per lo stesso hook, e puoi decidere l'ordine di priorità! Questo non si poteva con iptables.

#### Esempio: hooks (agganci nel Netfilter stack)

Quando crei una **base chain**, devi dire:

- **a quale hook agganciarla** (input, forward, output, prerouting, postrouting, o ingress)
- **la priorità**
- e la **policy** (accettare o droppare pacchetti se nessuna regola scatta)

Esempio semplice:

```
nft add table inet my_filter_table
nft add chain inet my_filter_table input_chain {
    type filter hook input priority 0;
    policy drop;
}
```

Questo crea una **base chain** chiamata `input_chain`, agganciata al **hook input**, con priorità 0 e policy predefinita: **drop tutto** se nessuna regola scatta.

### Tools e compatibilità

Per facilitare la transizione:

- Esistono **strumenti di conversione automatica** da iptables → nftables:
  - `iptables-translate`
  - `iptables-nft` (per usare ancora i vecchi comandi ma che girano con backend nftables)
- Ci sono **interfacce user-friendly**:
  - `UFW` (Uncomplicated Firewall, usato da Ubuntu)
  - `firewalld` (Usato da CentOS, Fedora, RHEL...)

Quindi se uno non vuole impararsi tutta la sintassi, può comunque **gestire il firewall in modo semplice**, oppure **mantenere la compatibilità con le vecchie regole**.

## IDS e IPS

Entrambi sono **sistemi di sicurezza** pensati per **proteggere reti e dispositivi da attacchi informatici**.

La **differenza storica** tra i due è:

Sigla	Nome completo	Cosa fa	Azione
<b>IDS</b>	Intrusion Detection System	Rileva un'intrusione in corso	Avvisa l'amministratore
<b>IPS</b>	Intrusion Prevention System	Previene un'intrusione	Blocca direttamente l'attacco

**Ma attenzione!**

Oggi questa distinzione non è più così netta: molti strumenti moderni fanno entrambe le cose.

Quindi spesso si parla di **IDPS** (Intrusion Detection and Prevention System).

### Tipi di IDS (*Intrusion Detection*)

#### 1. **NIDS – Network-based IDS**

- Posizionato in un punto strategico della rete (es. vicino a un router o switch).
- Analizza tutto il traffico in entrata/uscita nella sottorete.
- **Non modifica il traffico**, lo osserva soltanto e genera **allarmi** se rileva qualcosa di anomalo.
- Esempio: Snort, Suricata in modalità IDS.

#### 2. **HIDS – Host-based IDS**

- Installato direttamente su un host (es. server, PC).
- Analizza **log di sistema, modifiche ai file, traffico locale**.
- È utile anche per rilevare attacchi che *non passano dalla rete* (es. malware locali).

#### 3. **PIDS – Protocol-based IDS**

- Analizza il comportamento dei protocolli di rete (es. TCP, HTTP, FTP).

- Cerca comportamenti strani: pacchetti fuori ordine, flag anomali, etc.
- Molto utile per scovare attacchi più tecnici come quelli **a livello di protocollo**.

#### 4. APIDS – Application Protocol-Based IDS

- Specializzato nell'analizzare **protocolli di applicazione** (es. SQL, TDS, SIP).
- Capisce la logica del protocollo applicativo e **rileva anomalie semantiche** (es. query sospette verso un DB).

### Tipi di IPS (Intrusion Prevention)

#### 1. NIPS – Network-based IPS

- Come il NIDS, ma **può bloccare attivamente** il traffico sospetto.
- È posizionato **inline**, cioè *nel percorso del traffico*, e può scartare pacchetti in tempo reale.
- Es.: bloccare un attacco DoS o l'invio di exploit.

#### 2. HIPS – Host-based IPS

- Come l'HIDS, ma può **impedire esecuzioni sospette**, modifiche non autorizzate, ecc.
- È come un antivirus avanzato con capacità proattive.

#### 3. WIPS – Wireless-based IPS

- Monitora le **reti Wi-Fi**.
- Cerca dispositivi non autorizzati, access point rogue, attacchi tipo "Evil Twin" e può forzare la **disconnessione** (deauth).

#### 4. NBA – Network Behavior Analysis

- Analizza il comportamento del traffico in rete per cercare **pattern anomali**.
- Ad esempio: un host che inizia a fare tantissime connessioni TCP in pochi secondi → potrebbe essere un attacco.

### IDS vs IPS oggi

In teoria:

	IDS	IPS
Posizione	passiva (sniffer)	attiva (inline)
Azione	avverte	blocca
Rischio	nessuno (non modifica nulla)	può causare falsi positivi bloccando cose lecite

### Ma oggi, nella pratica:

- Tutti i sistemi **fanno detection e prevention** (sono IDPS).
- Gli strumenti più moderni **integrano capacità di apprendimento automatico**, euristiche, aggiornamenti automatici delle firme.
- Vengono spesso integrati direttamente nei **firewall**, nei **router**, o nelle **soluzioni cloud**.

### Esempi pratici

- **Snort**: IDS open-source (ma può essere anche configurato come IPS).
- **Suricata**: molto potente, può lavorare sia da IDS che da IPS.

Puoi pensare a IDS e IPS così:

- **IDS** = la videocamera che ti avvisa se vede un ladro
- **IPS** = il buttafuori che lo blocca all'ingresso

### Snort

**Snort** è uno dei software di sicurezza **IDS/IPS** più famosi e longevi, nato nel **1998**.

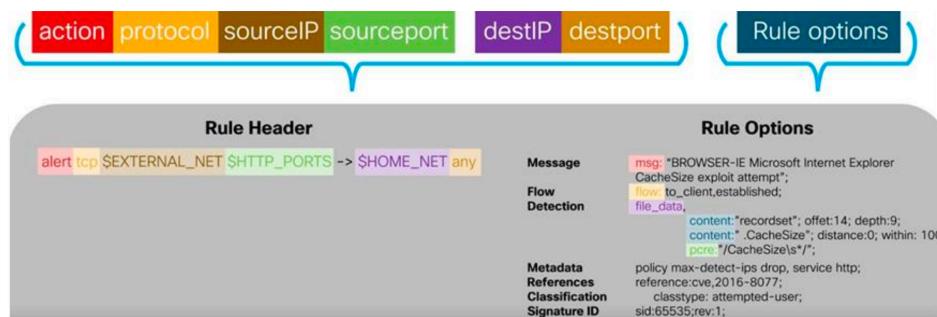
Dal **2013** è stato acquisito da **Cisco**, ma è rimasto **open source** e con una community molto attiva.

È classificato come un **NIPS** (Network-based Intrusion Prevention System), anche se può funzionare anche solo come **NIDS**.

### Snort 3: cosa è cambiato?

Snort ha avuto un'importante evoluzione con la **versione 3**:

- È **multi-threaded**, quindi riesce a sfruttare meglio le CPU moderne (prima usava solo un core).
- È **modulare e configurabile** via Lua.
- Ha **prestazioni simili** ai competitor in condizioni normali.
- **Sotto stress però perde pacchetti** (drop non intenzionali).
- Ha un **tasso di falsi positivi abbastanza alto**, quindi bisogna lavorare molto sulle regole.



### Cosa può fare Snort?

Snort lavora leggendo ogni pacchetto e decidendo **in base alle regole** cosa fare. **Azioni principali**:

Azione	Descrizione
<b>alert</b>	Genera un allarme e logga il pacchetto (senza bloccarlo)
<b>log</b>	Logga il pacchetto (più dettagliato di alert)
<b>drop</b>	Blocca il pacchetto e lo logga
<b>block</b>	Blocca <b>tutti</b> i pacchetti della connessione (compreso quello che ha causato l'allarme)
<b>pass</b>	Ignora il pacchetto (nessun log, nessuna azione)

### Azioni aggiuntive:

Azione	Descrizione
<b>react</b>	Risponde al client (es. con un messaggio) e chiude la connessione
<b>reject</b>	Interrompe la sessione ma <b>senza rispondere</b>
<b>rewrite</b>	Sovrscrive il contenuto del pacchetto (sconsigliato: meglio sanificare altrove)

**Nota: Riscrivere (rewrite) un pacchetto può rompere la comunicazione**

- Alterare il contenuto può causare:
  - **Errori di checksum**
  - **Problemi con l'ordine dei pacchetti**
  - **Incoerenze tra client e server**
- Rischi di compromettere la **sessione TCP** o rompere protocolli applicativi.
- **Snort rileva, non modifica**
- **Sanificare i contenuti spetta a strumenti specializzati**, come proxy o WAF, che lavorano più in alto nello stack (livello applicativo) e in modo più controllato.

### Come funziona una regola?

Le regole in Snort sono **testi strutturati** che specificano:

- l'**azione**
- il **protocollo**
- **ip e porta di origine e destinazione**
- e una **serie di opzioni** (content, metadata, reference, ecc.)

### Esempio di regola Snort:

```
alert tcp $EXTERNAL_NET any → $HOME_NET 21 (
    msg:"PROTOCOL-FTP serv-u directory traversal";
    flow:to_server,established;
    content:".%20.";
    fast_pattern;
    nocase;
    metadata:ruleset community;
    service:ftp;
    reference:bugtraq,2052;
    reference:cve,2001-0054;
    reference:nessus,10565;
    classtype:bad-unknown;
    sid:360;
    rev:16;
)
```

### Cosa fa questa regola?

- **Tipo:** `alert` → genera un allarme (non blocca)
- **Protocollo:** `tcp`
- **Da:** una sorgente esterna (`$EXTERNAL_NET any`)
- **Verso:** un server FTP interno (`$HOME_NET 21`)
- **Condizione:** nel payload deve esserci `".%20."` → *tentativo di directory traversal*
- **flow:** verso il server e connessione stabilita
- **fast\_pattern:** velocizza la ricerca del contenuto
- **nocase:** case insensitive
- **reference:** collegamenti a bug report, CVE e test di Nessus
- **sid:** ID della regola
- **rev:** versione della regola

### Snort vs WAF

- Snort (IPS) lavora a **livello di trasporto/rete**, fino al livello 4 → TCP/IP
- WAF (Web Application Firewall) lavora a **livello applicativo** → HTTP, SQL, ecc.

### Differenza cruciale:

- Snort non decifra il traffico HTTPS per **prestazioni**
- Il WAF invece **vede tutto già in chiaro** (deve farlo, è il suo lavoro!)

Per questo **alcune regole Snort si potrebbero implementare anche con un WAF**, soprattutto se parliamo di protezione da **attacchi applicativi** come XSS, SQLi, directory traversal ecc.

### Aggiornamenti e community

- Le regole di Snort sono **community-driven** (scritte da esperti, analisti, aziende).

- Gli aggiornamenti arrivano **molto spesso**, anche più volte a settimana.
- C'è un sistema di classificazione (sid, classtype, reference) per **organizzare e gestire** le regole.

## Suricata

**Suricata** è un **NIPS open-source**, nato nel **2010** e sviluppato da **OISF** (Open Information Security Foundation).

È un sistema di **Intrusion Detection & Prevention**, proprio come **Snort**, ma con qualche marcia in più su certi aspetti.

### Caratteristiche principali:

- **Multi-threaded** fin dalla nascita → sfrutta bene CPU multi-core
- Ottimo **sotto carico**: anche se consuma più RAM di Snort, **non perde pacchetti** facilmente
- Fa anche da **NSM (Network Security Monitoring)** → registra **tutto il traffico** che vede
- Salva log dettagliati in **formato JSON**, facilissimo da analizzare con strumenti come:
  - Elasticsearch
  - Kibana
  - ecc.

---

### Suricata vs Snort

Aspetto	Snort	Suricata
Multi-thread	Solo dalla v3	Sì, dalla prima versione
Precisione	Più falsi positivi	Più falsi negativi
Prestazioni sotto stress	Perde pacchetti	Più stabile
Logging avanzato	Solo log base	JSON + NSM completo
Parser Layer 7	Limitato	Avanzato (HTTP, TLS, DNS, SMTP, ecc.)
Protocol detection	Manuale	Automatica

In sintesi: Suricata è più **pesante**, ma anche più **completo** e **scalabile**.

---

### NSM (Network Security Monitoring)

Una delle **grandi differenze** tra Suricata e altri IPS è la capacità di fare **monitoraggio di rete avanzato**, cioè:

- Cattura e analizza tutto il traffico, non solo quello sospetto
- Ricostruisce le **sessioni** TCP, HTTP, TLS, DNS, SMTP...
- Salva **metadati strutturati** in JSON → ottimi per analytics, incident response, threat hunting

---

### Regole in Suricata

Le regole di Suricata sono **molti simili a quelle di Snort** (Suricata è compatibile con il formato delle regole Snort!), ma con alcune estensioni in più.

#### Esempio di regola:

```
alert http $HOME_NET any → $EXTERNAL_NET any (
  msg:"HTTP GET Request Containing Rule in URI";
  flow:established,to_server;
  http.method;
  content:"GET";
  http.uri;
  content:"rule";
  fast_pattern;
  classtype:bad-unknown;
  sid:123;
  rev:1;
)
```

### Cosa fa questa regola?

- **Tipo:** `alert` → genera un allarme
- **Protocollo:** `http`
- **Sorgente:** interna (`$HOME_NET any`)
- **Destinazione:** esterna (`$EXTERNAL_NET any`)
- **Condizioni:**
  - la connessione dev'essere stabilita e verso il server (`flow`)
  - deve esserci una **GET request** (`http.method` + `content:"GET"`)
  - nell'URI dev'esserci la parola "**rule**" (`http.uri` + `content:"rule"`)
- **Classtype:** tipo di attacco (`bad-unknown`)
- **sid/rev:** ID e versione della regola

### Note interessanti:

- Suricata ha un **parser HTTP** integrato, quindi puoi usare keyword come `http.uri`, `http.method`, `http.host`, ecc. in modo naturale.
- La regola può diventare **molto granulare**, proprio perché Suricata analizza fino al livello 7 (applicativo).

### Integrazione e Analisi

Grazie all'output in JSON e al supporto per **EVE JSON**, Suricata è perfetto per essere integrato in stack SIEM/monitoraggio:

Stack tipico:

- Suricata → genera log in JSON
- Filebeat → li spedisce a Elasticsearch
- Kibana → dashboard per visualizzare alert e traffico

## EDR (Endpoint Detection and Response)

L'**EDR** è un software pensato per **proteggere gli endpoint** (PC, laptop, server...) **aziendali**, andando **oltre i limiti dell'antivirus tradizionale**.

L'obiettivo è **rilevare, analizzare, rispondere** e persino **investigare** su minacce complesse, **in tempo reale** o quasi.

In pratica: mentre l'antivirus è più passivo e lavora a "firma", l'EDR è attivo, registra tutto e analizza continuamente il comportamento della macchina.

### Come funziona un EDR?

Un **EDR** ha in genere **5 funzioni fondamentali**:

#### 1. Raccolta dati continua

L'EDR installa un agente sull'endpoint che registra:

- Processi attivi/inattivi
- File creati, modificati o cancellati
- Connessioni di rete
- Siti visitati
- Comportamenti anomali dell'utente
- File trasferiti, dispositivi USB collegati, ecc.

Tutto questo viene raccolto e inviato a un server centrale per l'analisi.

#### 2. Analisi comportamentale e detection

L'EDR usa:

- **Rule-based detection** → regole fisse tipo "se vedo X, allora Y"
- **Machine learning** → per riconoscere comportamenti anomali (es. ransomware)
- **Threat intelligence** → confronta con indicatori noti

Riconosce **due tipi di segnali**:

#### **IOC – Indicator of Compromise**

Indizi di una **violazione avvenuta o in corso**.

Esempio:

"Connessione SSH da un IP noto per attività malevola"

#### **IOA – Indicator of Attack**

Indizi di un **comportamento anomalo o sospetto**, magari non ancora dannoso.

Esempio:

"10.000 file rinominati in `.crypt` in pochi secondi → sospetto ransomware"

---

### **3. Risposta automatizzata alla minaccia**

Appena l'EDR rileva qualcosa, può:

- **Generare alert** per il SOC (Security Operations Center)
- **Bloccare processi sospetti**
- **Revocare permessi** (lettura/scrittura/esecuzione)
- **Isolare il dispositivo** dalla rete
- **Disconnettere l'utente**
- **Attivare scansioni automatiche** su altri endpoint connessi

---

### **4. Isolamento e contenimento**

Se un attacco è in corso, l'EDR può:

- Mettere **offline il PC** senza spegnerlo (quindi si può continuare a fare analisi)
- Bloccare l'**utente sospetto**
- Limitare **accesso a file/cartelle di rete**
- Evitare **propagazione del malware**

---

### **5. Supporto all'analisi forense**

Uno dei **valori aggiuntivi più grossi** dell'EDR: ti aiuta a **capire cosa è successo**.

Può ricostruire:

- **Quali file sono stati compromessi**
- **Quali vulnerabilità sono state sfruttate**
- **Quali credenziali sono state usate o rubate**
- **Quando è partito l'attacco**
- **Come evitare che succeda di nuovo**

In sostanza, **aiuta il team di sicurezza** a fare "reverse engineering" dell'attacco e a **rafforzare le difese future**.

---

### **EDR vs Antivirus**

Aspetto	Antivirus	EDR
Firma	SI	SI: ma non solo
Analisi comportamentale	NO: o limitata	SI: avanzata
Machine Learning	NO	SI

Aspetto	Antivirus	EDR
Rilevamento exploit 0-day	Raro	Molto più probabile
Risposta attiva	NO	SI: (blocca, isola, ecc.)
Supporto a indagini forensi	NO	SI

### Esempio concreto

Immagina questo scenario:

1. Un utente apre una mail con un allegato Word malevolo.
2. Il file esegue uno script PowerShell in background.
3. Inizia la cifratura dei file (.crypt).

Un antivirus **forse** lo rileva.

Un EDR **sicuramente** nota:

- Che PowerShell è stato eseguito da Word (anomalia)
- Che decine di file vengono rinominati
- Che un processo tenta di connettersi a un dominio sospetto

E quindi **blocca tutto e isola l'host**, mentre avvisa il team.

## XDR (eXtended Detection and Response)

XDR sta per **Extended Detection and Response**, ed è l'**evoluzione naturale** dell'EDR.

Se l'EDR si concentra **solo sugli endpoint**, l'XDR **espande il raggio d'azione** per proteggere **tutti gli asset aziendali**:

- Endpoint
- Server (fisici e virtuali)
- Infrastruttura di rete
- Applicazioni
- Email
- Risorse **cloud**
- Container, workload, dispositivi IoT (in certi casi)

L'obiettivo è avere un'unica piattaforma centralizzata che rileva, analizza e risponde alle minacce in tutto l'ecosistema aziendale, e non solo sul PC della segretaria o del sysadmin.

### Differenza chiave: EDR vs XDR

Aspetto	EDR	XDR
Copertura	Solo endpoint	Endpoints + cloud + rete + email + app
Origine dati	Limitata (solo dispositivo)	Ampia (multi-layer)
Rilevamento minacce	Sul singolo host	Coordinato su più ambienti
Risposta	Locale (es. isolare host)	Globale (es. blocco su più asset)
Centralizzazione	Parziale	Totale, unico pannello di controllo
Integrazione con altri sistemi	Limitata	Alta (es. SIEM, SOAR, CASB...)

Metafora semplice:

- L'**EDR** è come una videocamera nel tuo ufficio: ti avvisa se succede qualcosa lì.
- L'**XDR** è come un sistema di videosorveglianza centralizzato per tutto l'edificio: reception, corridoi, garage, server room... tutto connesso.

### **Come funziona un XDR?**

Un sistema XDR ha **componenti sparsi** ovunque nella tua infrastruttura (on-premises e cloud), che raccolgono eventi e log.

Questi dati vengono:

1. **Centralizzati**
2. **Correlati tra loro**
3. **Analizzati** con tecniche avanzate (machine learning, regole, intelligence)
4. **Arricchiti** con contesto (da dove viene l'IP, quale utente è coinvolto, ecc.)
5. **Tradotti in azioni** (alert, isolamento, blocchi, risposta automatica...)

---

### **Fonti di dati che XDR può analizzare:**

- **Endpoint** → es. esecuzione anomala di PowerShell
- **Email** → es. allegato sospetto o link di phishing
- **Cloud** → es. accesso da IP anomalo a una VM in Azure
- **Rete** → es. scansioni o port knocking
- **Server** → es. comportamento sospetto di un servizio
- **Firewall / proxy / DNS** → es. contatti con domini malevoli

---

### **Rilevamento esteso**

L'XDR riesce a vedere **attacchi su più livelli** che magari sfuggirebbero all'EDR:

Esempio:

Un attaccante:

- Manda una mail di phishing (rilevato dal motore e-mail)
- L'utente clicca e scarica un eseguibile (rilevato dall'endpoint)
- Il malware si connette a un C2 in cloud (rilevato dal firewall o IDS)
- Si muove lateralmente nella rete (log di rete)

**L'XDR collega tutto questo** in una sola vista e può rispondere globalmente.

---

### **Risposta automatizzata (e orchestrata)**

L'XDR può rispondere **in maniera automatica e su più fronti contemporaneamente**:

- Isolare endpoint
- Bloccare IP e domini su firewall/DNS
- Disabilitare utenti su Azure AD
- Aggiornare policy su caselle email (es. blocco su Outlook)
- Inviare notifiche al SOC
- Attivare flussi di risposta (spesso integrati con SOAR)

---

### **Vantaggi principali dell'XDR**

**Panoramica completa** su tutta l'infrastruttura

**Migliore correlazione** tra eventi e alert più intelligenti

**Minor carico per gli analisti** (meno alert frammentati)

**Risposta automatizzata e integrata**

**Maggiore visibilità nel cloud**, sempre più usato

---

### Limiti (da conoscere)

L'XDR è potente, ma:

- È **più complesso** da gestire (soprattutto all'inizio)
- Richiede una **buona strategia di integrazione**
- Ha un **costo maggiore** rispetto all'EDR
- Se non correttamente configurato, **potrebbe generare troppi dati da gestire**

## Incident Response

L'**Incident Response (IR)** è l'**insieme di processi, procedure e strumenti** che un'organizzazione usa per:

- Rilevare una minaccia
- Analizzarla
- Contenerla
- Rispondere rapidamente
- Recuperare la normalità operativa

In pratica: quando succede un disastro (un attacco hacker, un ransomware, un furto di dati), l'**Incident Response** è il **piano d'azione che si attiva per mettere un freno al danno** e cercare di riprendere il controllo della situazione.

### Chi risponde agli incidenti? Il Blue Team!

Il **Blue Team** è il gruppo (interno o esterno) che si occupa della **difesa**.

Ma attenzione: il **Blue Team non "difende" in senso stretto, reagisce!** Questo perché:

- Gli attacchi spesso sono **nuovi e mai visti prima**
- Molti attacchi sono **personalizzati**, fatti su misura
- È **impossibile avere una difesa perfetta**: vulnerabilità, errori umani, nuovi malware... ce n'è sempre una nuova

Ecco perché si lavora in **modalità best effort**: si fa il massimo, ma l'**incidente può sempre accadere**.

---

### Definizione formale

"L'Incident Response è l'insieme di processi e tecnologie usati da un'organizzazione per individuare e rispondere a minacce informatiche, intrusioni o attacchi cibernetici."

---

### Tipi di incidenti comuni

#### 1. Ransomware

- Malware che **cripta i file** dell'hard disk e chiede un riscatto per sbloccarli.
- I primi ransomware erano più semplici: bloccavano l'avvio del sistema ma **potevi rimuovere l'hard disk e salvarti i file**.
- Oggi invece criptano tutto, rendendo i dati **illeggibili** senza la chiave (che solo l'attaccante possiede... forse).
- Pagare **non garantisce** il recupero.
- Di solito si consiglia: **ripristino da backup o formattazione totale**.

#### 2. Phishing

- Tecnica di ingegneria sociale dove l'attaccante **simula un messaggio legittimo** (email, SMS, WhatsApp...) per:
  - Farti cliccare su link malevoli
  - Scaricare malware
  - Inserire credenziali in un sito falso
  - Inviare denaro

Quando il phishing è **personalizzato** e mirato a una vittima specifica (es. il CEO), si parla di **spear-phishing**.

### 3. Social Engineering

Insieme di tecniche psicologiche per manipolare il comportamento umano e aggirare le difese tecniche.

Esempi comuni:

Tecnica	Descrizione
<b>Phishing</b>	Già visto sopra
<b>Baiting</b>	"Clicca qui per ricevere 500€" o "penna USB gratuita"...
<b>Tailgating</b>	Fisico: seguire qualcuno in un'area riservata. Digitale: usare un PC sbloccato
<b>Pretexting</b>	L'attaccante si finge tecnico che vuole "aiutare"
<b>Scareware</b>	"Il tuo PC è infetto! Scarica subito questo antivirus!"
<b>Quid pro quo</b>	Ti prometto qualcosa in cambio di un'azione (es. installare un programma)
<b>Watering hole</b>	L'attaccante compromette un sito che la vittima visita spesso (es. sito aziendale, forum tecnico ecc.)

#### Impatti possibili di un incidente

Vediamo ora **cosa può succedere di grave** in seguito a un attacco:

##### Perdite finanziarie

- Esempio: l'attaccante prende il controllo dell'email del CEO e manda richieste di pagamento ai dipendenti
- Oppure chiede riscatti (ransomware)

##### Problemi di business continuity

- Un attacco può mettere **fuori uso decine di sistemi** aziendali contemporaneamente
- Se l'attaccante ha fatto **movimento laterale**, pulire tutto è difficile
- In questi casi è spesso meglio **formattare e ripartire da backup puliti**

##### Esfiltazione di dati

- Furto di:
  - Documenti interni
  - Codice sorgente
  - Informazioni riservate su progetti o clienti
- Molto comune nei **data breach**

##### Perdita definitiva dei dati

- Ransomware distruttivo o attacchi deliberati
- Se **non ci sono backup**, si perdono per sempre

#### Perché è importante l'Incident Response?

Perché **nessun sistema è invulnerabile**.

Per quanto tu abbia firewall, antivirus, EDR, XDR, e chi più ne ha più ne metta... **prima o poi qualcosa passa**.

Quello che **fa la differenza** è **quanto sei pronto a reagire**:

- Hai un piano?
- Hai un team?
- Hai backup recenti?
- Sai chi chiamare? (es. CERT, forze dell'ordine, assicurazione, legale, PR)

## SIEM

**SIEM** sta per **Security Information and Event Management**

È un sistema che:

- **Raccoglie i log** da tantissime fonti (server, firewall, antivirus, applicazioni, ecc.)
- **Li conserva** in modo sicuro
- **Correla gli eventi** per individuare potenziali minacce
- **Lancia allarmi** se nota comportamenti sospetti

In altre parole, è il **cervello della sicurezza aziendale**: guarda tutto ciò che succede nella rete e cerca segnali di attacco.

---

### I log: la materia prima

Ogni sistema informatico (Windows, Linux, router, ecc.) **genera dei log**, cioè file dove viene scritto tutto quello che accade:

- Log di accesso
- Log di sistema
- Log di applicazioni
- Log di rete
- Eventi di sicurezza (login falliti, file modificati, ecc.)

Ma c'è un problema: **l'attaccante, una volta dentro, può cancellare o modificare i log!**

Ecco perché servono strumenti esterni e sicuri per proteggerli.

---

### Il ruolo del SIEM

Un SIEM viene installato su **una macchina separata e protetta**. Fa queste cose:

#### 1. Collezione i log

- Riceve log in tempo reale da tutti i sistemi della rete
- Più fonti → più visione globale

#### 2. Garantisce l'integrità dei log

- I log devono arrivare **subito** (idealmente **prima** di essere scritti sul disco locale!)
- Devono essere:
  - **Crittografati in transito** (es. TLS)
  - **Crittografati a riposo**
  - **Verificabili tramite hash crittografici** (meglio con *HMAC*)

Questo fa sì che **l'attaccante dovrebbe violare sia la macchina vittima che il SIEM** per nascondere le sue tracce.  
Difficile.

#### 3. Correla gli eventi

- Es: 10 tentativi falliti di login su 5 server in 30 secondi? → possibile brute-force.
- 1 utente effettua login da Italia, poi da Russia dopo 1 minuto? → possibile furto di credenziali.
- Tutti i log vengono **incrociati**, analizzati e **valutati con regole** preimpostate.

#### 4. Genera alert

- Quando succede qualcosa di anomalo, il SIEM **invia un avviso** al SOC (Security Operations Center)
- In alcuni casi può essere integrato con strumenti che reagiscono (come un SOAR)

---

### Requisiti fondamentali

Per funzionare correttamente, un SIEM deve rispettare alcuni **requisiti**:

Requisito	Perché è importante
Macchina separata	Così un attacco su una macchina non compromette il SIEM
Log inviati subito	Se arrivano tardi, l'attaccante ha tempo per cancellarli
Crittografia	Evita che i log vengano letti o modificati in transito o a riposo
Hash/HMAC	Verifica l'integrità dei log ricevuti

### Esempi di SIEM

Vediamo due esempi pratici:

#### IBM QRadar

- **SIEM commerciale** molto potente
- Fa anche:
  - EDR (protezione degli endpoint)
  - Log analysis
  - **SOAR** (Security Orchestration Automation and Response), soluzione simile a XDR che raccoglie eventi da vari endpoint, li correla e cerca di identificare e mitigare autonomamente le minacce. In generale è più complesso da gestire e configurare rispetto a XDR, e costa di più -- ma dipende dal SOAR utilizzato
  - Quindi si avvicina a un **XDR**

#### Wazuh (open-source)

- Più leggero e accessibile, molto usato nelle PMI (**Piccola e Media Impresa**)
- Ha 4 moduli principali:
  - **Indexer**: indice e conserva gli alert, allo scopo di poterli recuperare mediante ricerche rapide da parte dell'utente
  - **Server**: gestisce gli agent, ovvero li configura, li aggiorna e analizza i dati ricevuti per cercare indicatori di compromissione
  - **Dashboard**: interfaccia utente per la visualizzazione dei dati
  - **Agent**: componente installabile sull'endpoint da proteggere e monitorare

Wazuh è **estremamente flessibile** e integrabile con altre soluzioni.

### A cosa serve davvero un SIEM?

Il SIEM è **il cuore della visibilità e del controllo** di un'infrastruttura IT. Serve per:

- **Capire cosa succede**
- **Individuare attacchi in corso**
- **Supportare l'analisi forense** post-incidente
- **Essere compliant** (es. GDPR, ISO 27001, NIS2...)

### SOC

Il **Security Operations Center** è un **team specializzato** (non solo un luogo fisico!) che ha come missione:

- **Monitorare, analizzare e reagire** 24/7 a tutto ciò che succede nella rete aziendale
- **Gestire la sicurezza dell'intera infrastruttura IT**

In pratica, è **la centrale operativa della difesa informatica**. È come il 118 della cybersecurity: appena c'è un allarme, parte la risposta.

### Chi lo gestisce?

- A capo c'è di solito il **CISO** (Chief Information Security Officer) o un **SOC Manager**
- Il SOC può essere:
  - **Interno** (gestito dal personale dell'azienda stessa)

- **Esterno / MSSP** (affidato a **Managed Security Service Provider**, esperti esterni)

---

### Cosa fa davvero un SOC?

Attenzione: **non stanno tutto il giorno a guardare i log!**

Le attività sono tante e **molti dinamiche**. Vediamole per bene:

---

#### 1. Gestione dell'inventario degli asset

- Devono sapere **quali dispositivi esistono**, dove si trovano e se sono **protetti**
- Esempio: "Il server web ha un antivirus attivo? È aggiornato? Il backup funziona?"

---

#### 2. Manutenzione di routine

Attività periodiche, fondamentali per la prevenzione:

- **Aggiornamenti software e patching**
- Aggiornamento di:
  - Regole del **firewall**
  - Signature e policy di **IPS/IDS**
  - Regole del **SIEM**
- Verifica backup:
  - Si sono eseguiti correttamente?
  - I file di backup sono **leggibili e integri?**

---

#### 3. Incident Response

Quando qualcosa va storto, sono loro a intervenire subito!

- Segnalazione → Analisi → Contenimento → Risoluzione → Lezione appresa
- Gestione delle **procedure IR**, definite in anticipo
- **Simulazioni regolari**: chiamate anche **cyber exercises**, per essere sempre pronti

---

#### 4. Formazione continua

Il mondo cyber cambia ogni giorno. Il team SOC si tiene aggiornato su:

- **Nuove minacce**
- **Nuove tecnologie di difesa**
- Corsi, certificazioni, partecipazione a conferenze, ecc.

---

#### 5. Uso di strumenti avanzati

Oltre a firewall, IDS, SIEM ed EDR/XDR, usano:

- **Threat Intelligence Platform**: per sapere cosa succede nel mondo esterno
- **Honeypot**: trappole digitali per attrarre e studiare gli attaccanti
- **Sandbox**: per analizzare comportamenti sospetti in ambienti isolati

---

#### 6. Compliance

A volte il SOC si occupa anche di **verifica normativa**, cioè:

- Verificare che l'azienda sia **compliant** con leggi come:
  - GDPR
  - ISO 27001
  - NIS2
- Generare report di conformità
- Preparare audit di sicurezza

---

### **Collegamenti con altri sistemi**

Il SOC lavora **insieme** a strumenti come:

- **SIEM**: aggrega e analizza i log
- **EDR/XDR**: monitora gli endpoint/altri asset
- **SOAR**: automatizza la risposta agli incidenti
- **Firewall/IPS**: blocca attacchi in tempo reale

In pratica, il SOC **usa e orchestra tutti questi strumenti**.