Università degli studi di Perugia
Facoltà di Ingegneria

Appunti di Sicurezza Informatica

Alessio Burani
Silvia Cascianelli

Anno Accademico 2014/2015

Indice

Capitolo 1

Introduzione alla sicurezza informatica

1.1 Premessa

1.1.1 Definizione

Che cosa significa sicuro? La parola sicurezza e l'aggettivo sicuro vengono sempre associati a beni che si desidera proteggere da possibili danni, danneggiamenti, perdita, e via dicendo. In particolare un bene è al sicuro se è ben protetto, e la sua messa in sicurezza non deve impedirne l'utilizzo. Esistono molteplici definizioni di sicurezza informatica, ad esempio:

- Security is the degree of protection against danger, damage, loss, and criminal activity [**?**].
- Security is a form of protection where a separation is created between the assets and the threat [**?**].

La parola sicurezza deriva dal latino sine cura: senza preoccupazione. La sicurezza di un sistema può essere definita come la conoscenza del fatto che l'evoluzione del sistema non produrrà stati indesiderati. Le cause che possono minare la sicurezza sono molteplici e spesso non prevedibili, quindi non si può parlare di sicurezza in senso assoluto, ma solo relativo (*"L'unico computer sicuro è un computer spento"*).

Trasversalità della tematica: Le problematiche di sicurezza interessano molteplici campi (attività lavorative, vita domestica, hobby, giochi, sport, etc.). Di fatto, ogni settore della vita moderna ha delle implicazioni relative alla sicurezza. Il livello di sicurezza di un'organizzazione dipende dai livelli di sicurezza di tutti i suoi comparti/settori. Il livello di sicurezza di un sistema è determinato dal livello di sicurezza dal suo comparto meno sicuro (principio dell'anello debole).

1.2 Concetti fondamentali

1.2.1 Sistema informativo e informatico

- Per sistema informativo (information system) di un'organizzazione si intende l'insieme delle informazioni prodotte ed elaborate e delle risorse umane, materiali e immateriali, coinvolte nel processo di elaborazione di tali informazioni
- Per sistema informatico (information and communication technology system) s'intende l'insieme delle varie tecnologie coinvolte nel sistema informativo (il sistema informativo è parte del sistema informatico).

Questo corso verterà sulla sicurezza dei sistemi informativi. Tuttavia, si approfondiranno maggiormente questioni inerenti la sicurezza dei sistemi informatici. Per sicurezza di un sistema

informativo si intende il grado di protezione contro qualsiasi minaccia ai suoi asset. Richiede il soddisfacimento dei seguenti requisiti:

- **confidenzialità, integrità e disponibilità**
- **assicurazione, autenticità e anonimato**

Quando si parla di **attacco** solitamente si intende la violazione di uno o più di questi requisiti.

1.2.2 Vulnerabilità, Minacce, Attacchi, Difesa

Vulnerabilità

Una vulnerabilità (o falla o breccia) è una **debolezza intrinseca** di un sistema, che potrebbe essere sfruttata per provocare perdite o danni. Scaturisce spesso da errate procedure di sicurezza e/o da errori di progettazione/implementazione, e in alcuni casi è intimamente legata alla natura del sistema. Ad esempio:

- un sistema potrebbe essere vulnerabile alla manipolazione non autorizzata dei dati causa un bug nella procedura di autenticazione dell'utente
- un calcolatore è vulnerabile all'acqua

Nel primo caso (vulnerabilità scaturite da errate procedure di sicurezza) l'insorgenza delle vulnerabilità può essere mitigata adottando adeguati standard e norme di qualità.

Minacce (Threats)

Per minaccia (threat) ad un sistema informatico/informativo si intende quell'insieme di circostanze che potrebbero arrecare danni ai suoi asset: eventi potenziali, accidentali o deliberati, che, nel caso accadessero, produrrebbero perdite e danni. il realizzarsi di una minaccia generalmente avviene sfruttando una o più vulnerabilità del sistema. Si parla quindi di situazioni ipotetiche che potrebbero avvenire in determinate circostanze. Ad esempio:

- esecuzione di codice malevole che invia dati sensibili ad un'organizzazione criminale
- accesso a dati riservati da parte di entità non autorizzate
- perdita di dati a causa della rottura di un apparato hardware o al crash di uno specifico software

Attacchi (Attacks)

Un attacco (attack) è un atto deliberato teso ad arrecare un danno al sistema. Consiste, di fatto, nella realizzazione di una **minaccia**. Generalmente, un attacco viene perpetrato attraverso lo sfruttamento di una o più vulnerabilità. Spesso si classificano in base all'entità del danno:

- **attacco attivo (active attack)**: altera le risorse o ne modifica il processo di gestione/elaborazione
- **attacco passivo (passive attack)**: ottiene le informazioni/dati senza alterarli e senza modificare il relativo processo di gestione/elaborazione

Un'altra importante classificazione è in base "al luogo" da cui viene iniziato l'attacco:

- **attacco dall'interno (inside attack)**: attacco iniziato da un'entità all'interno del perimetro di sicurezza di un sistema informativo di una data organizzazione
- **attacco dall'esterno (outside attack)**: attacco iniziato da un'entità all'esterno del perimetro di sicurezza

Ovviamente, è molto più difficile prevenire e rilevare gli attacchi interni di quelli esterni. Ciò anche a causa del fatto che le misure di prevenzione per questo tipo di attacchi limita notevolmente l'usabilità del sistema (si pensi, ad esempio, alla struttura gerachica in ambiente militare, in cui ogni risorsa conosce il minimo indispensabile per svolgere i propri compiti. In questo modo nel caso la risorsa venga compromessa, si limita il danno. Ovviamente tutto ciò rallenta il processo di funzionamento del sistema).

Tecniche di difesa

Diverse contromisure (o misure protettive) possono essere attuate per proteggere un sistema informativo da eventi accidentali e da attacchi deliberati. Tali misure devono essere strutturate all'interno di un piano di sicurezza redatto dopo un'attenta analisi costi/benefici (textitcost-effective solutions). Le tecniche di difesa possono essere di tipo:

- **preventivo**: effettuano una serie di **controlli** per evitare **a priori** che attacchi noti o immaginabili possano essere sferrati con successo (e.g. controlli aeroportuali, controllo di accessi e permessi negli OS).
- **a posteriori**: sono tese a ridurre gli effetti di un attacco che è riuscito a eludere le misure preventive di cui sopra; devono monitorare un sistema ed essere in grado di **rivelare** comportamenti anomali.

Un **meccanismo di sicurezza** è un qualsiasi metodo, strumento, o procedura teso a rilevare, prevenire o porre rimedio agli effetti di un attacco alla sicurezza del sistema. La strategia di difesa, qualunque essa sia, combina in modo opportuno uno o più meccanismi di sicurezza. molti meccanismi di sicurezza consistono in controlli hardware/software.

1.2.3 Sicurezza informatica: definizione classica e requisiti CIA

La sicurezza informatica si fonda sulla protezione dei seguenti macro-requisiti di un sistema informativo (informatico):

- **Confidenzialità (Confidentiality)**
- **Integrità (Integrity)**
- **Disponibilità (Availability)**

Spesso si utilizza l'acronimo C.I.A. per denotarli in modo compatto. Ovviamente, come si può

Figura 1.1: Diagramma di Venn dei macro-requisiti di sicurezza

notare dal diagramma di Venn in Figura 1.1 spesso vengono richiesti contemporaneamente più di questi requisiti, cosi come un attacco può violare più requisiti.

Confidenzialità (Confidentiality)

Def. **Confidentiality**: the property that information is not made available or disclosed to unauthorized individuals, entities, or processes. Per confidenzialità si intende quindi la garanzia che alle risorse informatiche accedano solo le parti autorizzate ad accedervi. E' talvolta denominata segretezza, riservatezza o privacy. Nota bene: per accesso non si intende solo la lettura, ma anche la visualizzazione, la stampa o la semplice consapevolezza dell'esistenza di una data risorsa nel sistema.

Attacchi alla confidenzialità: Si ha un attacco alla confidenzialità quando una entità (persona, processo o risorsa) tenta di accedere senza autorizzazione a informazioni protette (mentre queste sono memorizzate, oppure durante l'elaborazione, oppure ancora durante una comunicazione). La protezione della confidenzialità avviene utilizzando in modo appropriato i seguenti strumenti (meccanismi) di sicurezza informatica:

- **Cifratura (Encryption)**
- **Controllo degli accessi (Access Control)**
- **Sicurezza fisica (Physical security)**

Integrità (Integrity)

Def. **Integrity**: safeguarding the accuracy and completeness of information and processing methods.
Def. **Integrity**: the property of safeguarding the accuracy and completeness of assets.

Per integrità si intende quindi la garanzia che le risorse possano essere modificate solo dalle parti autorizzate e solo nei modi prestabiliti. Le modifiche comprendono la scrittura, la variazione, il cambiamento dello stato, l'eliminazione e la creazione. Nel caso di file, conviene includere anche i metadati associati (proprietario, ultimo utente ad averlo letto, data ultima modifica,data di creazione), in modo che un accesso non autorizzato al contenuto possa essere rivelato da un controllo di integrità applicato ai metadati.

Attacchi all'integrità: Si ha un attacco all'integrità quando una entità (persona, processo o risorsa) tenta di modificare senza autorizzazione una o più risorse del sistema informativo.

La protezione della confidenzialità avviene utilizzando in modo appropriato i seguenti strumenti (meccanismi) di sicurezza informatica (tutti basati su un uso corretto della ridondanza):

- **Backup**
- **Somma di controllo o Checksum**
- **Codici a correzione di errore (Corruzioni non deliberate ma accidentali, possono essere facilmente elusi da attacchi intelligenti)**
- **Codici di autenticazione dei messaggi o Message Authentication Code MAC**

Disponibilità (Availability)

Def. **Availability**: ensuring that authorized users have access to information and associated assets when required.

Per disponibilità (availability) si intende quindi che le risorse siano accessibili, nei tempi e nei modi prestabiliti, alle parti autorizzate ogni volta che le richiedono: se una persona o un sistema dispone dei diritti di accesso ad una risorsa, l'accesso non deve essergli impedito. Spesso la disponibilità viene citata tramite il suo opposto: la **negazione di servizio** (**Denial of Service** o **DoS**). La disponibilità può assumere significati/sfumature diverse; una risorsa può trovarsi in uno stato intermedio tra i due opposti stati di piena disponibilità e di piena indisponibilità.

Conflittualità requisiti CIA

Spesso i requisiti di confidenzialità, integrità e disponibilità possono essere in conflitto tra loro. E' quindi importante trovare il compromesso ottimo per la garanzia di tutti e tre. E' facile garantire la confidenzialità di una risorsa impedendo a chiunque di accedervi (e.g. se chiudo la risorsa in un blocco di cemento, sicuramente resterà confidenziale. Tuttavia la disponibilità della stessa verrebbe compromessa in senso assoluto).

1.2.4 Requisiti AAA

In aggiunta ai concetti CIA, sovente viene richiesto il soddisfacimento di ulteriori requisiti che vanno sotto l'acronimo A.A.A.

Assicurazione (Assurance)

Per assicurazione (assurance) si intende come viene fornita e gestita la fiducia reciproca tra le varie parti coinvolte nel sistema informativo. Tale requisito sopperisce la mancanza di regolamentazione, da parte dei requisiti C.I.A., dell'uso delle risorse di un sistema. E' teso quindi ad evitare un uso non consono dei vari asset del sistema. Tale concetto è ovviamente bidirezionale: così come il sistema deve fidarsi degli utenti (del fatto che non lo usino in modo inappropriato), gli utenti devono fidarsi del sistema (e.g.:trattamento dei dati personali). Il concetto di fiducia è tuttavia difficile da quantificare ed è legato al grado di confidenza sul comportamento che ci si attende dal sistema. Il requisito di assicurazione viene regolato agendo sui seguenti strumenti:

- **Politiche (Policies):** specifiche comportamentali che regolano l'operato degli attori del sistema
- **Permessi (Permissions):** descrivono le operazioni ammesse/concesse e quelle proibite
- **Protezioni (Protections):** implementazione dei meccanismi di sicurezza tesi ad applicare e far rispettare le politiche e i permessi di cui sopra
- **Qualità del software:** lo sviluppo del software che rispetta standard di qualità rigorosi rende più difficile che il sistema si discosti dai comportamenti attesi

Autenticità (Authenticity)

L'autenticità è la capacità di provare che dichiarazioni, politiche e autorizzazioni rilasciate da persone o sistemi siano veritiere. Se non è garantita persone/sistemi possono sostenere argomentazioni non vere senza essere contraddetti da prove oggettive. Se è garantita, si dice anche che è soddisfatto il requisito del textbfnon-ripudio (non-ripudiation). Per non-ripudio si intende che dichiarazioni autentiche rilasciate da persone e/o sistemi NON possono essere negate. La firma digitale è il principale meccanismo crittografico che permette di ottenerlo. Un contesto in cui tale requisito è importante è, ad esempio, la PEC.

Anonimato (Anonymity)

Per anonimato (anonymity) si intende che non è possibile ottenere o risalire all'dentità di un individuo pur avendo accesso a determinati record/transazioni di un sistema informativo. Se un'organizzazione deve rendere pubblici alcuni dati dei suoi clienti/membri senza violarne la privacy, può adottare alcuni dei seguenti strumenti:

- **Aggregazione (Aggregation):** combinare mediante somme e/o medie i dati di diversi individui
- **Miscelazione (Mixing):** permutare i dati dei singoli utenti in modo da preservare l'esito di predeterminate query
- **Mediazione (Proxies):** utilizzare degli agenti fidati che si interpongono tra l'individuo e i sistemi con i quali interagisce

- **Pseudonimi (Pseudonyms):** utilizzare delle identità fittizie eventualmente autenticate da una terza entità che funge da garante

1.2.5 Tipologie di Minacce e Attacchi

E' possibile classificare diverse minacce e attacchi rispetto a quali requisiti violano. Vediamone alcune:

Intercettazione (Eavesdropping)

Si ha un'intercettazione quando un'entità non autorizzata ha ottenuto l'accesso ad una risorsa. Generalmente questo tipo di attacchi avviene nella fase di trasmissione dell'informazione, e rappresenta una violazione alla confidenzialità. Esempi:

- copia illecita di file di programma o dati
- intercettazione di una comunicazione telefonica
- sniffing di pacchetti di dati

Un'intercettazione potrebbe essere molto difficile da rilevare, poiché un intercettatore "silenzioso" potrebbe essere molto abile e non lasciare tracce o concellarle.

Alterazione (Alteration)

Si ha una alterazione o modifica quando un'entità non autorizzata, oltre ad accedere a una risorsa, interferisce con essa modificandola. Rappresenta una violazione alla confidenzialità. Esempi:

- attacchi di tipo **man-in-the-middle**: un flusso di dati viene intercettato, modificato e ritrasmesso (rappresenta anche una minaccia alla confidenzialità)
- virus informatici che modificano file di sistema critici in modo da eseguire azioni maliziose
- cambiare i valori di un database o modificare un programma in modo che esegua dei calcoli diversi

Alcuni casi di alterazione possono essere facilmente rilevati, mentre altre modifiche, più sottili, possono essere estremamente difficili da individuare.

Interruzione (Denial-of-service)

In un'interruzione, una risorsa del sistema viene degradata o eliminata, diventando non disponibile o inutilizzabile. Rappresenta una minaccia alla disponibilità e/o all'integrità. Esempi:

- la distruzione o il sabotaggio di un dispositivo
- la cancellazione di un file
- la congestione di un Web server causata da un numero enorme di richieste artificiali

Falsificazione (Masquerading)

La falsificazione consiste nella contraffazione di risorse (dati/hardware) da parti non autorizzate. Costituisce principalmente una violazione di autenticità, e a volte anche alla confidenzialità e/o all'integrità. Esempi:

- phishing: creazione di siti Web apparentemente identici agli originali allo scopo di frodarne gli utenti
- textbfspoofing: spedizione di pacchetti dati con falsi indirizzi di ritorno

A volte le falsificazioni sono facilmente rilevabili, ma se attuate con abilità potrebbero essere del tutto indistinguibili da normali operazioni reali legittime.

Ripudio (Ripudiation)

Il ripudio consiste nel negare di aver effettuato una data azione. Ll'azione potrebbe essere la ricezione o la spedizione di un messaggio, così come l'esecuzione di una transazione. Spesso, riguarda il tentativo di recedere da un contratto (accordo) precedentemente assunto in cui era comunque previsto l'uso di ricevute (o simili) per dimostrare l'esecuzione di determinate operazioni. Si tratta di un attacco all'assicurazione.

Inferenza (Inference), Correlation and traceback

Per inferenza o attacco inferenziale si intendono un insieme di tecniche che utilizzando la statistica e l'algebra permettono di ricavare/stimare **informazioni sensibili** a partire da dati non sensibili. Rappresenta un attacco alla confidenzialità, e spesso è attuato nel contesto dei databases. Similmente, per correlation and traceback si intendono un insieme di tecniche basate sulla statistica e sull'algebra che permettono di determinare la sorgente di una particolare informazione o di un particolare flusso di dati.

N.B.: C'è una differenza, in senso giuridico, tra dati personali e dati sensibili. Tale distinzione dipende dalla giurisdizione dei paesi. In particolare in Italia il dato personale viene indicato come un'informazione che permette di identificare un individuo (anagrafica), mentre un dato sensibile rappresenta un'informazione su aspetti della vita privata dell'individuo (orientamento sessuale, opinione politica, credo religioso, etc.)

1.2.6 Principi della Sicurezza Informatica

Mediazione completa (Complete mediation):

Ogni accesso ad una risorsa deve essere controllato verificando che sia conforme alle politiche di sicurezza stabilite; diffidare da miglioramenti nell'efficienza ottenuti salvando autorizzazioni precedentemente acquisite, poiché i permessi possono variare nel tempo.

Struttura aperta (Open design):

L'architettura, il progetto e l'implementazione dei meccanismi di sicurezza di un sistema devono essere resi pubblici.

- la sicurezza deve fondarsi sulla segretezza di pochi elementi chiave
- maggior feedback favoriscono l'individuazione di bug, falle e vulnerabilità, aumentando la robustezza e la sicurezza del sistema
- un meccanismo di protezione ritenuto sicuro da molti è preferibile ad uno noto solo a pochi. E' quindi bene evitare meccanismi di sicurezza basati sulla segretezza (security by obscurity).

Separazione dei privilegi (Separation of privilege):

Più condizioni dovrebbero essere richieste per concedere l'accesso a risorse limitate o ottenere il permesso di effettuare una data azione. In genere questo principio comporta una separazione logico/funzionale delle componenti di un sistema.

Minimo privilegio (Least privilege):

Ogni parte di un sistema deve avere i privilegi minimi necessari allo svolgimento dei propri compiti. Per attività inusuali che richiedono maggiori privilegi conviene assegnare autorizzazioni temporanee fortemente limitate nel tempo. In questo modo si riduce il rischio di attacchi basati sulla scalata di privilegi.

Minimo meccanismo comune (Least common mechanism):

I meccanismi di sicurezza che per l'accesso e la gestione di risorse condivise non dovrebbero essere a loro volta condivisi o dovrebbero essere condivisi il meno possibile. E' quindi buona norma adottare tecniche di isolamento quali la virtualizzazione e il sandboxing (e.g. browser web: nessuna applicazione web può agire sul filesystem, eccetto attraverso i cookies). In questo modo vengono mitigati rischi derivanti da comportamenti malevoli di utenti cui spetta comunque l'accesso a una data risorsa condivisa.

Usabilità (Usability, Psychological acceptability):

I meccanismi di sicurezza non devono rendere più difficile l'accesso alle risorse. Le interfacce utente devono essere ben progettate, intuitive e di facile utilizzo, mentre i parametri di configurazioni inerenti aspetti di sicurezza devono essere di semplice comprensione e facilmente modificabili.

Fattore lavoro (Work factor):

Il costo necessario ad aggirare un meccanismo di sicurezza deve essere confrontabile alle risorse di cui dispone un potenziale attaccante. Un meccanismo di sicurezza deve avere un livello di sofisticazione, e pertanto un costo, che tenga conto del valore degli asset da proteggere e delle risorse a disposizione di potenziali attaccanti.

Monitoraggio (Compromise recording):

A volte può convenire effettuare un monitoraggio dettagliato piuttosto che investire in sofisticati meccanismi di sicurezza di tipo preventivo.

Penetrazione più semplice:

un attaccante utilizzerà qualsiasi mezzo di penetrazione disponibile: non necessariamente i mezzi più ovvi (prevedibili), non necessariamente i mezzi per i quali sono state installate le difese più solide. (e.g.: E' inutile avere un sistema informatico sicuro se il personale non viene formato e fornisce a chiunque credenziali di accesso). L'applicazione di tale principio presenta le seguenti difficoltà:

- saper anticipare l'avversario, cioè riuscire a prevederlo
- gestire in modo equilibrato la sicurezza delle diverse parti di un sistema; rafforzare le difese di una parte può indurre gli avversari ad attaccare un'altra parte (più debole) del sistema.

Temporalità:

le risorse di un sistema informativo devono essere protette solo fino a quando possiedono un valore, e in modo proporzionale al loro valore. Generalmente tale principio si riferisce ai dati: il loro valore può subire brusche variazioni; si consideri ad esempio il valore dei dati sull'andamento dei mercati prima e dopo la loro divulgazione.

Anello più debole:

la sicurezza di un sistema articolato NON può essere più forte del suo anello più debole. La gestione della sicurezza deve tener conto del sistema nel suo insieme. Un elevato grado di sicurezza delle singole parti non implica un elevato grado di sicurezza globale, ma è necessario prevedere anche una strategia oculata di coordinamento della varie parti che non introduca vulnerabilità (l'insieme è più della somma delle parti).

1.3 Fondamenti di Crittografia

La parola Crittografia deriva dal greco, e significa "scrittura segreta". La crittografia è quindi l'arte e la scienza dello scrivere in modo segreto, ovvero l'arte di offuscare le informazioni in modo incomprensibile prevedendo una tecnica segreta per ricostruirle in modo esatto. L'applicazione storica della crittografia è la protezione della confidenzialità dei messaggi. Con **crittografia classica** si intende, infatti, l'insieme di tecniche che sopperiscono la necessità di nascondere il contenuto di una comunicazione a soggetti non autorizzati. Con l'avvento dei calcolatori e del digitale, si è passati alla crittografia moderna, che presenta molte applicazioni aggiuntive di non immediata comprensione, ma estremamente utili:

- firma digitale
- controllo di integrità sicuro
- autenticazione

1.3.1 Terminologia e definizioni preliminari

- **Testo in chiaro (plaintext o cleartext):** messaggio nella sua forma originale
- **Testo cifrato (ciphertext):** messaggio cifrato (criptato o crittografato)
- **Cifratura o crittazione o criptazione (encryption):** processo che produce il testo cifrato a partire dal testo in chiaro
- **Decifratura o decrittazione o decriptazione (decryption):** processo inverso della cifratura
- **Crittografo (cryptographer):** esperto di crittografia
- **Crittoanalisi o crittanalisi (cryptanalysis):** l'arte o la scienza di violare testi cifrati; ricostruire il testo in chiaro senza disporre del segreto necessario alla fase di decifratura
- **Crittoanalista:** esperto di crittoanalisi
- **Crittologia (cryptology):** l'arte o la scienza delle scritture nascoste nella sua accezione pi'u generale; include la crittografia e la crittoanalisi
- **Chiave segreta:** la segretezza dell'elaborazione nell'algoritmo è di solito concentrata nella segretezza di una chiave. E' stata introdotta perché è difficile concepire ogni volta nuovi algoritmi di cifratura ed è difficile spiegare rapidamente il funzionamento di un nuovo algoritmo ad un soggetto con il quale si desidera comunicare in modo sicuro

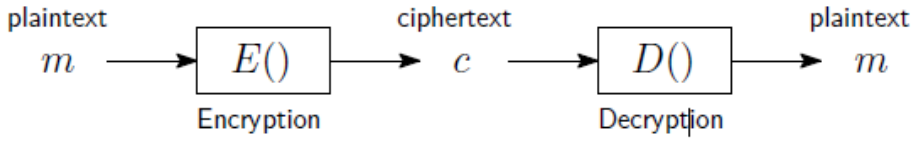


Figura 1.2: Schema a blocchi che illustra il processo cifratura-decifratura

1.3.2 Definizione formale di un sistema crittografico

: Formalmente, un sistema crittografico è costituito da sette componenti (a volte alcune di queste sono assenti o coincidono):

- l'insieme dei possibili input (plaintext)

- l'insieme dei possibili output (ciphertex)
- l'insieme delle possibili chiavi di cifratura
- l'insieme delle possibili chiavi di decifratura
- la corrispondenza tra chiavi di cifratura e di decifratura
- l'algoritmo di cifratura da usare
- l'algoritmo di decifratura da usare

1.3.3 Complessità computazionale

Uno schema crittografico anche assumendo che sia privo di vulnerabilità intrinseche, può essere (quasi) sempre sottoposto ad attacchi a forza bruta. Affinché ciò possa avvenire è necessario disporre di un test affidabile che permetta di dire se una chiave scelta a caso è quella corretta; cioè, essere in grado di riconoscere il testo in chiaro. Se tale precondizione è verificata si può perseguire un approccio esaustivo in cui si tentano tutte le chiavi. Uno schema crittografico è allora tanto più sicuro quanto è maggiore il costo computazionale necessario a violarlo. Contemporaneamente deve garantire l'efficienza di cifratura/decifratura. Formalizzando, possiamo definire un sistema **computazionalmente sicuro** se:

- il costo per rendere inefficace il cifrario supera il valore dell'informazione cifrata
- il tempo richiesto per rendere inefficace il cifrario supera l'arco temporale in cui l'informazione ha una qualche utilità

Tuttavia, stimare lo sforzo computazionale richiesto per effettuare con successo la crittoanalisi del testo cifrato è molto difficile. Il tempo richiesto per un approccio a forza bruta fornisce soltanto un limite superiore: è realistico solo se l'algoritmo non presenta delle debolezze intrinseche di tipo matematico; in questo caso si possono effettuare stime ragionevoli su tempi e costi. In un approccio a forza bruta mediamente si devono provare metà di tutte le possibili chiavi. Ovviamente in questo processo prende parte, come variabile fondamentale, la lunghezza della chiave. Alcuni schemi crittografici prevedono una chiave avente lunghezza variabile: aumentandola aumenta la sicurezza, ma diminuisce l'efficienza. Altri schemi stabiliscono a priori la lunghezza della chiave, se si desidera aumentarla è necessario sviluppare algoritmi simili che utilizzano chiavi di lunghezza diversa, oppure è possibile combinare in modo opportuno tali schemi ottenendo uno schema risultante con una chiave globale più lunga (attenzione al modo in cui si combinano!).

1.3.4 Algoritmo pubblico o segreto?

Ottenere la sicurezza dalla segretezza cioè custodendo gelosamente e nascondendo la metodologia di cifratura/decifratura, **security by obscurity**, è assolutamente da evitare. Questo perché:

- mantenere la segretezza è difficile, poiché le tecniche di reverse engineering permettono di risalire al codice
- è una strategia in chiara opposizione al fundamental tenet of cryptography

Oggigiorno, nelle applicazioni di uso civile/commerciale si usano schemi crittografici di dominio pubblico. La segretezza, laddove prevista, è conseguenza di altre cause (e.g. protezione del segreto industriale, contesti militari).

1.3.5 Attacchi a sistemi crittografici

Si possono distinguere cinque categorie di attacchi in base al tipo di informazioni in possesso dell'attaccante (di cui gli ultimi due meno frequenti):

Solo testo cifrato (ciphertext only)

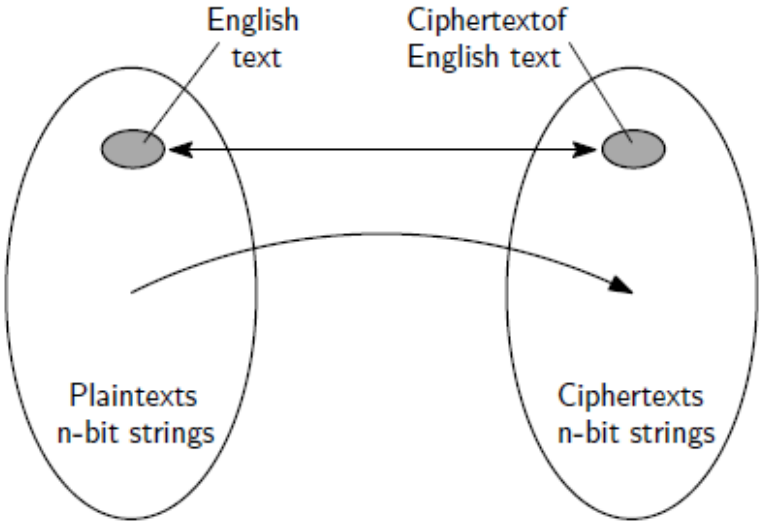
L'attaccante conosce l'algoritmo di cifratura e dispone soltanto di una certa quantità di testo cifrato. Nell'ipotesi che non vi siano vulnerabilità intrinseche nello schema crittografico, l'attaccante può tentare un attacco forza bruta, a patto che disponga di un test affidabile per il riconoscimento del testo in chiaro e di potenza di calcolo sufficiente. Tale attacco è anche noto come **testo in chiaro riconoscibile (recognizable plaintext)**. Riguardo la riconoscibilità del testo in chiaro valgono le seguenti osservazioni:

- nel caso di messaggi testuali, è molto improbabile che una chiave di decifratura errata permetta di ottenere un messaggio verosimile
- è essenziale avere una sufficiente quantit'a di testo in chiaro

Esempio: il messaggio m è una frase di un qualche linguaggio naturale. Ipotizzando che i caratteri di m siano codificati in ASCII a 8 bit e il testo cifrato c abbia la stessa lunghezza di m (quasi sempre è così). Siano:

- t: il numero di caratteri di m e di c
- $n = 8t$: il numero di bit di m e di c
- $2^{\alpha n}$: il numero totali di messaggi distinti, nel linguaggio naturale considerato, di lunghezza n bit

Si osservi che nominalmente esistono 2^n stringhe binarie di n bit, ma solo una piccolissima frazione di queste è la codifica di un messaggio nel linguaggio naturale, $0 < \alpha < 1$ è un fattore correttivo usato per modellare tale fenomeno, per la lingua inglese $\alpha = 0.16$.



Il seguente rapporto:

$$\frac{2^{\alpha n}}{2^n} = \frac{1}{2^{(1-\alpha)n}} \tag{1.1}$$

rappresenta la probabilità che un stringa random di n bit costituisca la codifica di un messaggio nel linguaggio naturale considerato. Se la chiave di decifratura ha una lunghezza di k bit, effettuando un attacco a forza bruta (2^k tentativi), il numero atteso di messaggi verosimili (nel linguaggio naturale dato) associabili al testo cifrato c è pari a:

$$\frac{2^k}{2^{(1-\alpha)n}} \tag{1.2}$$

Essendo k fissato, al crescere di n tale numero tende rapidamente a zero, quindi il test di riconoscibilità diventa quasi infallibile se si dispone di una sufficiente quantità di testo cifrato. In alcuni casi, si potrebbe tentare un approccio a forza bruta computazionalmente meno costoso se:

- la chiave di decifratura coincide con quella di cifratura, spesso è così
- la chiave di cifratura 'e derivata da una password di utente segreta con una procedura nota (si può tentare un approccio a forza bruta sullo spazio delle password, sensibilmente più piccolo dello spazio delle chiavi)

IMPORTANTE: uno schema crittografico deve SEMPRE essere sicuro contro attacchi di tipo **ciphertext only**, poiché il testo cifrato è sempre facilmente intercettabile e ottenibile.

Testo in chiaro conosciuto (known plaintext)

In questo caso, il crittoanalista conosce:

- l'algoritmo di cifratura
- un certo insieme di coppie $\langle plaintext, ciphertext \rangle$, ma non ha la facoltà di decidere lui il tipo specifico di plaintext
- una certa quantità di testo cifrato

Alcuni schemi crittografici potrebbero essere molto resistenti ad attacchi di tipo **ciphertext only** ma rivelarsi vulnerabili ad attacchi di tipo **known plaintext**. In caso di impiego, è fondamentale prevenire che un attaccante ottenga coppie $\langle plaintext, ciphertext \rangle$.

Testo in chiaro selezionato (chosen plaintext)

In questo caso, il crittoanalista conosce:

- l'algoritmo di cifratura
- una certa quantità di testo cifrato (con chiave segreta K)

e può scegliere a piacimento il testo in chiaro ed ottenere il relativo testo cifrato (sempre con la chiave K). In altre parole può scegliere quali coppie $\langle plaintext, ciphertext \rangle$ conoscere. E' possibile che un sistema crittografico sia sicuro contro attacchi di tipo **ciphertext only** e **known plaintext**, ma sia vulnerabile ad attacchi di tipo **chosen plaintext**.

Testo cifrato selezionato (chosen ciphertext)

L'attaccante dispone (oltre che dell'algoritmo di cifratura e del testo cifrato da decifrare) di testo cifrato, con significato, scelto da lui, e corrispondente testo in chiaro.

Testo selezionato (chosen text)

L'attaccante dispone sia di testo in chiaro scelto da lui (e corrispondente testo cifrato) sia di testo cifrato scelto da lui (e corrispondente testo in chiaro).

1.3.6 Tipi di funzioni crittografiche

- funzioni a chiave pubblica (public key functions): richiedono l'uso di due chiavi, una pubblica e una privata (approfodire non ripudio su firma digitale)
- funzioni a chiave segreta (secret key functions): richiedono l'uso di una singola chiave
- funzioni hash (hash functions): non usano alcuna chiave

Tipologia di attacco	Informazioni in possesso al crittoanalista
solo testo cifrato (*ciphertext only*)	• algoritmo di cifratura • testo cifrato da decifrare
testo in chiaro conosciuto (*known plaintext*)	• algoritmo di cifratura • testo cifrato da decifrare • uno o più testi in chiaro e corrispondenti testi cifrati
testo in chiaro selezionato (*chosen plaintext*)	• algoritmo di cifratura • testo cifrato da decifrare • testo in chiaro scelto dal crittoanalista e relativo testo cifrato
testo cifrato selezionato (*chosen ciphertext*)	• algoritmo di cifratura • testo cifrato da decifrare • testo cifrato, con significato, scelto dal crittoanalista e corrispondente testo in chiaro
testo selezionato (*chosen text*)	• algoritmo di cifratura • testo cifrato da decifrare • testo in chiaro scelto dal crittoanalista e relativo testo cifrato • testo cifrato, con significato, scelto dal crittoanalista e corrispondente testo in chiaro

Figura 1.3: Tabella riassuntiva tipologie di attacchi

Key Size (bits)	Number of Alternative Keys	Time required at 1 decryption/µs	Time required at 10^6 decryptions/µs
32	$2^{32} = 4.3 \times 10^9$	2^{31} µs = 35.8 minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	2^{55} µs = 1142 years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	2^{127} µs = 5.4×10^{24} years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	2^{167} µs = 5.9×10^{36} years	5.9×10^{30} years
26 characters (permutation)	$26! = 4 \times 10^{26}$	2×10^{26} µs = 6.4×10^{12} years	6.4×10^6 years

Figura 1.4: Tempo medio per una ricerca esaustiva

1.4 Modelli di controllo degli accessi (Access control models)

Il controllo degli accessi alla varie risorse di un sistema informativo costituisce un fondamentale meccanismo di sicurezza di tipo **preventivo**. Previene attacchi alla confidenzialità, all'integrità e all'anonimato. L'idea di base è restringere l'accesso solo a coloro che hanno la necessità di accedere e/o modificare specifiche risorse, in accordo al principio del **minimo privilegio**.

1.4.1 Matrice di controllo degli accessi

Una matrice di controllo degli accessi (Access Control Matrix ACM) è una tabella che definisce i permessi di accesso dei vari **soggetti** di un sistema informativo ai suoi **oggetti**.

- un **soggetto** è un utente, un gruppo o una generica entità attiva che desidera effettuare una data azione su un data risorsa
- un **oggetto** è un file, un documento, un dispositivo, una generica risorsa o più in generale un'entità passiva sulla quale si desidera compiere una data azione

Ogni riga della tabella è associata ad un soggetto, e ogni colonna è associata ad un oggetto. Ogni cella stabilisce quindi il tipo di azione consentita; il soggetto e l'oggetto sono implicitamente definiti dalla cella. Diverse modalità di accesso sono possibili: lettura, scrittura, copia, esecuzione, cancellazione, annotazione. Una cella vuota stabilisce che non viene assegnato alcun tipo di accesso.

	/etc/passwd	/home/mario/	/admin/
root	read, write	read, write, exec	read, write, exec
pippo	read		
mario	read	read, write, exec	
backup	read	read, exec	read, exec
...	...	...	...

Figura 1.5: Esempio access control matrix

Pro e Contro ACM

L'adozione di una ACM offre, in linea di principio, i seguenti vantaggi:

- immediatezza nel valutare e modificare i diritti di accesso per una data coppia soggetto-oggetto
- facilità di visualizzazione e gestione: semplifica la vita all'amministratore

Purtroppo, il grande svantaggio delle ACM, che ne limita fortemente l'applicabilità, è la mancanza di scalabilità. E' del tutto ragionevole pensare che un computer server possa avere ordine di 10^3 soggetti (per lo più utenti) e 10^6 oggetti (files e directories), richiedendo una ACM di 10^9 celle. In questi ordini di grandezza i precedenti vantaggi decadono (non scalabilità).

1.4.2 Liste di controllo degli accessi (Access Control List ACL)

Una lista di controllo degli accessi (access control list ACL) segue un approccio di tipo "object-centered" per garantire una buona scalabilità, in particolare:

- ad ogni oggetto **o** viene associata un lista **L**, detta la lista di accesso di **o**, che enumera tutti i soggetti che hanno un qualche diritto di accesso ad **o**, e

- per ciascuno di tali soggetti, **s**, sono specificati i tipi di azioni che **s** può compiere su textbfo

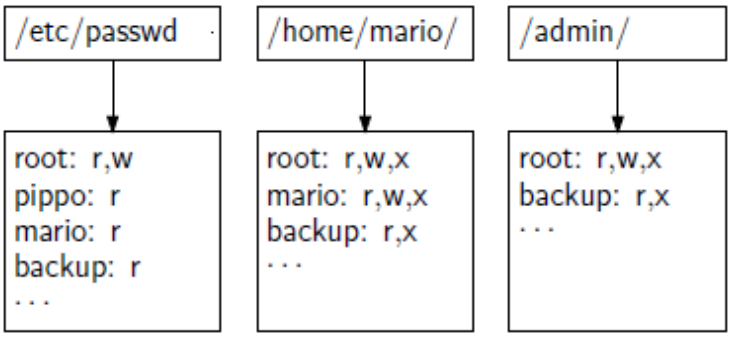


Figura 1.6: Esempio access control list

Nella pratica, il modello ACL comporta un significativo risparmio di memoria rispetto al modello ACM: ogni lista del modello ACL è ottenibile scandendo la relativa colonna del modello ACM ignorando però le celle vuote.

Pro e Contro ACL

Il principale vantaggio del modello ACL rispetto al modello ACM è la minor occupazione di memoria che si registra nella pratica (in teoria, nel caso peggiore l'occupazione è la medesima). La memoria occupata complessivamente nel modello ACL, $\sum_O size(L_O)$, è proporzionale a $C_{nv}(ACM)$ cioè al numero di celle non vuote della ACM. Nella pratica $C_{nv}(ACM)$ è molto minore di $C(ACM)$ (che denota il numero totale di celle della ACM); cioè nella pratica $C_{nv}(ACM) << C(ACM)$. Un altro vantaggio è la facilità di gestione delle ACL da parte del sistema operativo:

- sono incorporate nei metadati associati all'oggetto in questione (i.e. filesystem)
- per verificare i diritti di accesso ad un oggetto o non è necessario consultare una struttura dati centralizzata associata a tutti gli oggetti
- ma basta manipolare una struttura dati molto più snella, distribuita ed associata soltanto all'oggetto **o**

Il principale svantaggio delle ACL è che non consente di enumerare in modo efficiente i diritti di accesso di un dato soggetto. Tale operazione deve essere espletata ogni qual volta un utente viene rimosso da un sistema. I diritti di accesso di un soggetto **s** possono ottenersi soltanto effettuando una scansione di tutte le liste di accesso (associate a tutti gli oggetti **o**), e selezionando i diritti di accesso di **s** nelle liste in cui tale soggetto compare. Nel modello ACM, invece, ciò poteva banalmente ottenersi esaminando la riga della matrice relativa al soggetto **s**.

1.4.3 Controllod egli accessi basato su liste di capacità (Capabilities Access Control CAC)

Il modello basato sulle liste di capacità segue un approccio "subject-centered", complementare (ortogonale) a quello del modello ACL, per offrire una buona scalabilità. Ad ogni soggetto **s** viene associata una lista, detta **C-list** di **s**, contenente soltato gli oggetti per i quali **s** ha un qualche diritto di accesso. Per ciascun oggetto **o** della **C-list** di **s** viene specificato il tipo di azione che **s** può esercitare su **o**. Similmente al modello ACL, anche il modello CAC comporta un significativo risparmio di memoria rispetto al modello ACM nelle situazioni pratiche: ogni lista del modello CAC è ottenibile scandendo la relativa riga del modello ACM ignorando però le celle vuote.

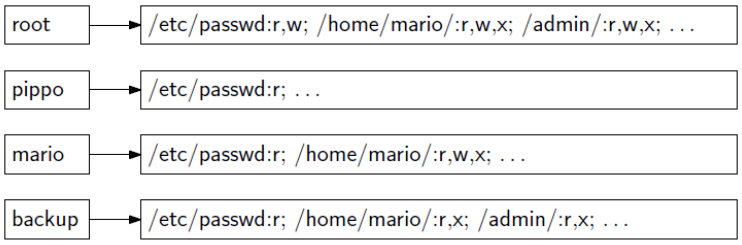


Figura 1.7: Esempio CAC

Pro e Contro CAC

Rispetto al modello ACM, anche il modello CAC richiede una minor occupazione di memoria nella pratica. la memoria occupata complessivamente nel modello CAC, $\sum_s size(L_s)$, è proporzionale a $C_{nv}(ACM)$, cioè al numero di celle non vuote della ACM (vedi considerazioni su ACL). Facilita inoltre il compito dell'amministratore di sistema nell'analizzare e gestire i diritti di accesso di un dato soggetto. Inoltre, l'autorizzazione ad accedere ad un dato oggetto o da parte di un soggetto **s** richiede tempi ragionevolmente brevi qualora la **C-list** di **s** abbia una dimensione contenuta.

Il principale svantaggio del modello CAC è che le **C-list** non sono direttamente associate agli oggetti. Ciò comporta che:

- non è possibile implementarle in modo distribuito incorporandole nei metadati degli oggetti
- l'unico modo per determinare chi e come può accedere ad un dato oggetto o è effettuare una ricerca su tutte le C-list (di tutti i soggetti); nel modello ACM tale operazione richiede semplicemente di esaminare la colonna relativa ad o (problema ortogonale a quello delle ACL)

1.4.4 Controllod egli accessi basato sui ruoli (Role-Based Access Control RBAC)

Per controllo degli accessi basato sui ruoli si intende che l'assegnazione dei diritti di accesso avviene in modo indiretto ed è funzione del ruolo attribuito ad un dato soggetto **s**. L'amministratore definisce i ruoli e specifica i diritti di accesso per tali ruoli, anziché quelli per i singoli soggetti (utenti). Ogni ruolo dovrebbe rappresentare una classe di soggetti (utenti) con la medesima mansione. I soggetti vanno assegnati ai vari ruoli coerentemente alle loro mansioni, e un soggetto può ricevere più ruoli dato che può ricoprire più mansioni. Definiti i ruoli, i diritti di accesso vanno assegnati secondo una logica ruolo-oggetto e NON soggetto-oggetto. I diritti di accesso di un dato soggetto sono l'unione dei diritti di accesso dei suoi ruoli. Pertanto:

- textbfil modello RBAC non 'e alternativo ai modelli ACM, ACL e CAC, ma si pone ad un livello di astrazione superiore: RBAC deve necessariamente sfruttare un modello per il controllo degli accessi, basato su **ACM**,**ACL** o **CAC**, ove i soggetti saranno però sostituiti dai ruoli.
- RBAC deve inoltre mantenere/gestire l'elenco dei ruoli associati ai vari soggetti

Architettura RBAC

Il modello RBAC 'e realizzabile utilizzando un generico framework per la gestione del controllo degli accessi (ACM, ACL, CAC), ove i soggetti sono, come già detto, sostituiti con i ruoli, più un layer superiore che gestisce l'elenco dei ruoli associati ai vari soggetti e che traduce le richieste di accesso per un dato soggetto in quelle relative ai suoi ruoli.

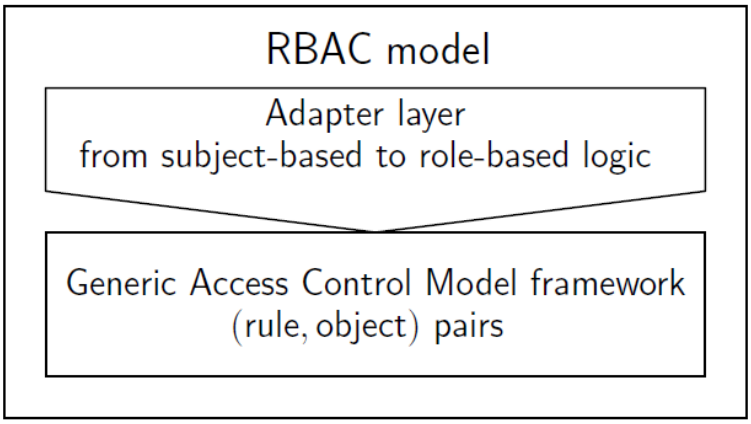


Figura 1.8: Architettura RBAC

Gerarchia dei ruoli

Generalmente è conveniente organizzare i ruoli in una struttura gerarchica, così da riflettere l'organigramma (gerarchico) di una data organizzazione. I diritti di accesso vengono più agevolmente gestiti e assegnati ai vari ruoli sfruttando l'ereditarietà.

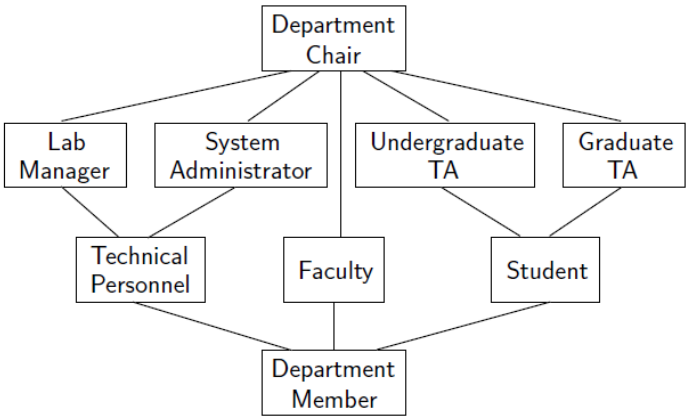


Figura 1.9: Architettura RBAC

Pro e Contro RBAC

Indipendentemente dal framework di basso di livello utilizzato per il controllo degli accessi, i vantaggi nell'uso del modello RBAC sono:

- la riduzione drastica del numero totale di regole di accesso da gestire; nella pratica il numero dei ruoli è significativamente inferiore a quello dei soggetti, inoltre la loro organizzazione gerarchica ne semplifica ulteriormente la gestione.
- l'overhead per determinare se un dato soggetto ha un dato diritto è contenuto, è sufficiente consultare se almeno uno dei suoi ruoli ha tale diritto.

Lo svantaggio principale è che non viene implementato dagli attuali sistemi operativi!

Capitolo 2

Secret Key Cryptography

2.1 Introduzione

La crittografia a chiave segreta richiede l'uso di UNA sola chiave: dato un messaggio (il testo in chiaro) e la chiave, la cifratura produce dati non intellegibili (il testo cifrato). Il testo cifrato ha circa la stessa lunghezza di quello in chiaro e la decifratura è l'inverso della cifratura, ed usa la stessa chiave. La crittografia a chiave segreta è talvolta chiamata crittografia **convenzionale** o crittografia **simmetrica**.

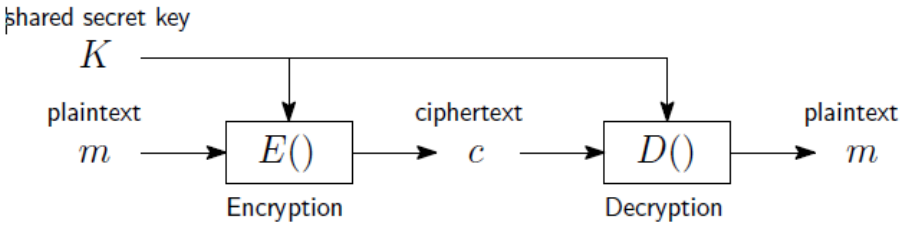


Figura 2.1: Schema a blocchi crittografia a chiave segreta

2.1.1 Impieghi crittografia a chiave segreta

La crittografia a chiave segreta è alla base di molti meccanismi di sicurezza, i suoi principali impieghi sono:

- protezione della **confidenzialità** (l'uso classico di queste tecniche è rappresentato dalle **comunicazioni su un canale insicuro**, uno degli usi più moderni invece dalla **memorizzazione sicura su supporto insicuro**)
- protezione dell'**integrità**: tramite queste tecniche possono essere eseguiti test per rilevare eventuali modifiche non consentite.
- autenticazione: tramite l'implementazione di protocolli per verificare l'identità di persone e/o processi.

Comunicazioni su un canale insicuro

In molte circostanze, due entità devono comunicare attraverso un canale insicuro correndo il rischio di essere ascoltate da una terza parte. Questo contesto è molto comune: si pensi alle reti LAN, che trasmettono dati in broadcast. La crittografia a chiave segreta permette a due entità che condividono un segreto (la chiave) di comunicare attraverso un canale insicuro, ove non può

essere garantita l'assenza di intercettazioni/ascoltatori (eavesdropper), avendo la garanzia che il contenuto della comunicazione rimarrà confidenziale.

Memorizzazione sicura

Si supponga di disporre di un supporto di memorizzazione non protetto (ad esempio accessibile a molti utenti). Se si desidera salvare i dati proteggendone la confidenzialità si può definire una chiave segreta, salvare i dati dopo averli crittografati con tale chiave e custodire la chiave segreta in un luogo protetto. Il rischio dell'uso di questa tecnica consiste nella possibilità di smarrimento della chiave. In tal caso i dati sarebbero irrevocabilmente persi.

Autenticazione forte

Per **autenticazione forte (strong authentication)** si intende che si è in grado di provare la conoscenza di un segreto, che contraddistingue l'identità di una data entità, senza rivelarlo. L'autenticazione forte è ottenibile utilizzando la crittografia a chiave segreta, ed è particolarmente utile quando due processi devono comunicare su una rete insicura. A rigore il segreto che contraddistingue l'identità che si desidera autenticare dovrebbe essere noto solo a quest'ultima. Nella crittografia chiave segreta tale requisito non può essere soddisfatto. Il segreto in questione è una chiave crittografica che deve essere nota anche all'entità autenticante.

Esempio

Si supponga che Alice e Bob condividano una chiave segreta K_{AB} e che vogliano autenticarsi reciprocamente, cioè ciascuno vuole accertarsi dell'identità dell'altro.
ipotesi: K_{AB} è nota solo ad Alice e Bob. Alice deve dimostrare a Bob di conoscere K_{AB} senza rivelarla e viceversa.
Strategia a sfida e risposta: ciascuno dimostra di conoscere K_{AB} rispondendo ad una sfida posta dall'altro. La **sfida** è un numero/stringa random **r** non prevedibile e sempre diversa. La **risposta** alla sfida è la sfida stessa cifrata $E(K_{AB}, R)$. In Figura 2.2 è riportato un possibile schema di autenticazione a sfida e risposta (challenge-response) a chiave segreta. La procedura segue i seguenti passi:

- Alice genera un numero random r_A (la sfida) e la invia al presunto Bob
- il presunto Bob critta la sfida con la sua chiave segreta K'_{AB} e restituisce ad Alice la risposta $E(K'_{AB}, r_A)$
- Alice riceve la risposta del presunto Bob e la decritta con la chiave K_{AB}, cioè calcola $D(K_{AB}, E(K'_{AB}, r_A))$. Se ottiene r_A, allora il presunto Bob è realmente Bob poiché con elevatissima probabilità, se $D(K_{AB}, E(K'_{AB}, r_A)) = r_A$, allora $K'_{AB} = K_{AB}$. In caso negativo deduce invece che il presunto Bob è un impostore. In modo analogo Bob verifica l'identità di Alice.

La sicurezza del precedente protocollo si fonda sulle seguenti condizioni che non devono venire meno:

- solo e soltanto Alice e Bob devono conoscere la chiave segreta K_{AB}
- le sfide generate devono essere **randomiche**, e di conseguenza non prevedibili, e **non ripetibili**, cioè la probabilità che due sfide si ripetano deve tendere a zero. L'attaccante potrebbe infatti collezionare molte coppie testo in chiaro/testo cifrato.
- è importante quindi che il numero di bit di una sfida sia superiore ad una data soglia (almeno 64 bit)

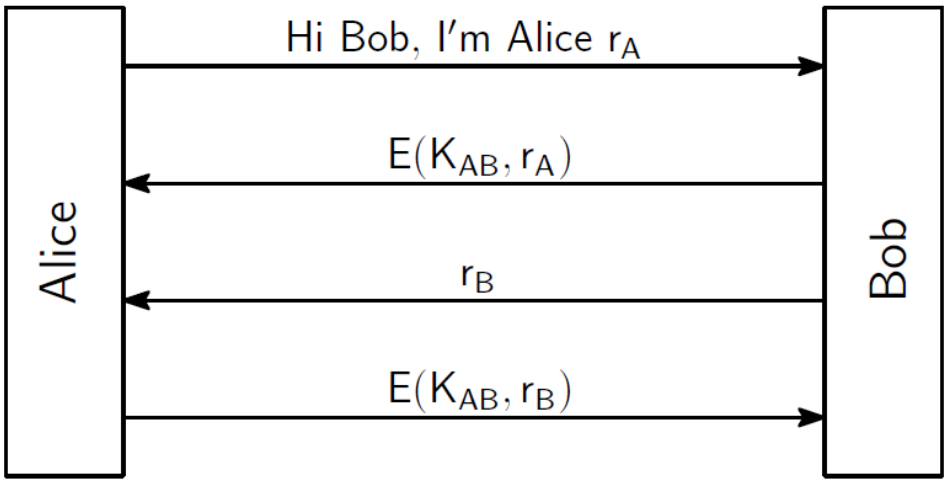


Figura 2.2: Schema a blocchi crittografia a chiave segreta

Controllo di integrità

La verifica dell'integrità di un messaggio inviato o di un file è un problema ricorrente in telecomunicazioni e in informatica. Per rilevare eventuali modifiche accidentali si fa uso generalmente di codici (somme) di controllo, detti anche **checksum**. Un **checksum** associa ad un qualsiasi messaggio $\mathbf{m} \in \{0,1\}^*$ un codice di lunghezza prefissato di b bit (generalmente $b = 32, 64, 128$ bit): $ck(\mathbf{m}) \in \{0,1\}^b$. Un buon **checksum** dovrebbe variare in modo significativo anche a fronte di iminime variazioni dell'input.

La sorgente del messaggio m_s rende pubblico/invia il corrispondente checksum $ck(m_s)$. Chi riceve il messaggio m_r calcola il checksum e verifica se vale l'uguaglianza $ck(m_s) = ck(m_r)$. In caso affermativo, conclude che $m_s = m_r$. Si noti tuttavia che, seppur improbabile, è possibile ottenere dei falsi positivi, cioè $ck(m_s) = ck(m_r)$ anche se non è vero.

Checksum segreti e non segreti

I codici di controllo servono per proteggere l'hardware da difetti e da inevitabili errori/guasti. Esistono codici di controllo molto sofisticati come i **CRC (Cyclic Redundancy Check)** per i quali la probabilità di falsi positivi è estremamente ridotta. Esistono anche i codici **FEC (Forward Error Correction)** che permettono di correggere eventuali errori oltre che a rilevarli (aggiungendo ulteriore ridondanza). Tuttavia entrambe queste tecniche **non** sono utilizzabili per la protezione contro attacchi intelligenti. Essendo pubblici, infatti, un avversario intelligente che vuole cambiare un messagio potrebbe modificare anche il codice di controllo in maniera coerente.

Per la protezione contro modifiche maliziose ad un messaggio, è richiesto un codice di controllo (checksum) segreto. Se l'algoritmo non è noto, nessuno può calcolare il checksum corretto per il messaggio modificato. Chiaramente, come nel caso degli algoritmi di cifratura, anziché un algoritmo segreto conviene avere un algoritmo noto a tutti che richiede la conoscenza di una chiave segreta per il calcolo di un codice di controllo (Vedi pagina 10, principio **Open Structure**).

In ciò consiste appunto un checksum cifrato, detto anche **MIC (Message Integrity Code)**. Il funzionamento del MIC è il seguente:

- l'algoritmo produce un codice di autenticazione di lunghezza fissa $MIC(K, m)$, denominato anche **MAC (Message Authentication Code)**
- il codice MIC MIC(K,m) viene trasmesso insieme al messaggio m stesso

- formalmente l'input è una coppia $(K, m) \in \{0,1\}^k \times \{0,1\}^*$, dove k denota il numero di bit della chiave segreta K, mentre l'output è sempre una stringa binaria di lunghezza prefissata, cioè $MIC(K, m) \in \{0,1\}^b$

2.1.2 Cifrari a blocchi e cifrari a flusso

Un **cifrario a blocchi** elabora un blocco di elementi in ingresso per volta, producendo un blocco di uscita per ciascun blocco di ingresso. Questa metodologia implica quindi che:

- il testo in chiaro deve essere preliminarmente suddiviso in blocchi
- il testo cifrato si ottiene combinando i vari blocchi cifrati

DES, IDEA e AES sono esempi di cifrari a blocchi simmetrici. Un **cifrario a flusso**, invece, elabora continuamente gli elementi in ingresso, producendo in uscita un "flusso" di elementi cifrati. Gli elementi cifrati vengono prodotti singolarmente, uno alla volta, man mano che la cifratura procede.

2.2 Cifratura a blocchi

2.2.1 Introduzione e concetti generali

L'algoritmo di cifratura converte un blocco di testo in chiaro in un blocco di testo cifrato. La chiave K non deve essere troppo corta (e.g. se K ha lunghezza 4 bit, sono sufficienti $2^4 = 16$ tentativi per individuarla). Analogamente la lunghezza (fissata) di un blocco non deve essere troppo piccola (e.g. se un blocco ha lunghezza 8 bit, ottenendo delle coppie plaintext - ciphertext si potrebbe costruire una tabella di $2^8 = 256$ coppie utilizzabile per la decfiratura).
D'altra parte, avere blocchi esageratamente lunghi oltre a non essere necessario dal punto di vista della sicurezza, comporta una gestione più complicata e può degradare le prestazioni. **64 bit è una lunghezza ragionevole per un blocco**:

- è improbabile ottenere ordine di 2^{64} coppie $\langle plaintext, ciphertext \rangle$ per costruire una tabella di decifratura, e
- anche se fosse possibile, la sua memorizzazione richiederebbe una spazio enorme (2^{64} record da 64 bit),
- come pure l'ordinamento per consentire ricerche efficienti

Il modo più generale per cifrare un blocco da 64 bit è definire una **biiezione** $\gamma : \{0,1\}^{64} \to \{0,1\}^{64}$. Tuttavia, memorizzare la definizione della *biiezione* in una struttura dati è impraticabile: sarebbero richiesti $2^8 4 \times 64 = 2^{70}$ bit. Ad essere precisi ce ne vogliono un po' di meno, trattandosi di un **biiezione**, comunque almeno 2^{69} bit sono richiesti (perché?!?!). Inoltre, così facendo la chiave è incorporata nella biiezione: per renderla parametrica rispetto alla chiave è necessario memorizzare una biiezione per ogni possibile chiave.

I sistemi di crittografia a chiave segreta sono concepiti per usare una chiave ragionevolmente lunga (ad esempio 64 bit e generare una **biiezione** che appare, a chi non conosce la chiave, completamente random. Se la biiezione fosse realmente random due input i e i' nei quali cambia un solo bit (uno qualunque) sarebbero **statisticamente indipendenti**: non può succedere, ad esempio, che il terzo bit dell'output cambia **sempre** quando il dodicesimo bit dell'input cambia. Gli algoritmi crittografici sono pensati per **diffondere/spargere** in tutti i bit dell'output il valore di ogni bit dell'input: si cerca di far si che ogni bit dell'output dipenda allo stesso modo da tutti i bit dell'input.

2.2.2 Sostituzione e Permutazione

Sostituzioni e permutazioni sono due trasformazioni base applicabili ad un blocco di dati.

Si assuma di dover cifrare un blocco di k bit. Una **sostituzione** specifica, per ciascuno dei 2^k possibili valori dell'input, i k bit dell'output. Per specificare una sostituzione **"completamente random"** sono necessari circa $k2^k$ bit. E' quindi impraticabile implementare una sostituzione per blocchi di 64 bit, mentre è fattibile per blocchi di lunghezza di 8 bit. Esempio di crittografia per sostituzione è il **Cifrario di Cesare**.

Una **permutazione** specifica, per ciascuna delle k posizioni dei bit in input, la posizione del corrispondente bit nell'output. Per specificare una permutazione **"completamente random"** per un blocco di lunghezza k bit sono necessari $k \log_2 k$ bit. Infatti per ciascuno dei k bit va specificata la sua posizione nell'output; ogni posizione richiede $k \log_2 k$. Ad esempio: essendo $2^6 = 64$, sono necessari $6 = \log_2 64$ bit per specificare la nuova posizione che l'i-esimo bit in input avrà in output.

Si noti che una **permutazione** è un caso particolare di **sostituzione** in cui ogni bit dell'output ottiene il suo valore da esattamente un bit dell'input.

2.2.3 Cifrario a blocchi – schema generale

Un algoritmo di cifratura a chiave segreta può funzionare come segue:

- scompone il blocco in input in pezzi più piccoli (e.g. blocchi da 8 bit)
- applica una sostituzione (tramite una **rete combinatoria**) a ciascun pezzo da 8 bit (la sostituzione dipenderà dal valore della chiave)
- gli output delle sostituzioni vengono riuniti in un unico blocco (64 bit)
- tale blocco viene permutato in un permutatore a 64 bit (che ha il compito di diffondere le modifiche eseguite nelle sostituzioni)
- il processo viene ripetuto un certo numero di volte riportando l'output in ingresso

2.2.4 Cifrario a blocchi – esempio

Ogni attraversamento del cifrario viene detto **round**. In riferimento alla Figura 2.3 si fanno le seguenti considerazioni. Con un solo round, un bit b_x di input può influenzare soltanto 8 bit $b_{x1}, b_{x2}, \ldots, b_{x8}$ dell'output, poiché b_x ha attraversato soltanto un blocco di sostituzione. In generale i bit $b_{x1}, b_{x2}, \ldots, b_{x8}$ non sono consecutivi essendo stati mescolati nel permutatore. Alla fine del secondo round, assumendo che i bit $b_{x1}, b_{x2}, \ldots, b_{x8}$ siano smistati in blocchi di sostituzione distinti, il bit b_x iniziale influenza tutti i bit in output.

2.3 Data Encryption Standard - DES

DES fu pubblicato nel 1977 dal **National Bureau of Standards**, ora rinominato **N**ational **I**nstitute of **S**tandards and **T**echnology (**NIST**), per usi commerciali e "altre" applicazioni del governo statunitense. Progettato da IBM, si basa sul precedente cifrario *Lucifer* ed è frutto della collaborazione con consulenti della NSA. DES usa una chiave di 56 bit, e mappa un blocco di input da 64 bit in un blocco di output da 64 bit (l'algoritmo è quindi abbastanza rigido, al contrario di altri cifrari a blocchi). La **chiave** è in realtà costituita da una sequenza di 64 bit, ma un bit in ogni ottetto (il blocco da 64 bit è formato da 8 sequenze di 8 bit dette ottetti) è usato come **odd parity** su ciascun ottetto. Di fatto, soltanto 7 bit in ogni ottetto sono quindi veramente significativi come chiave.

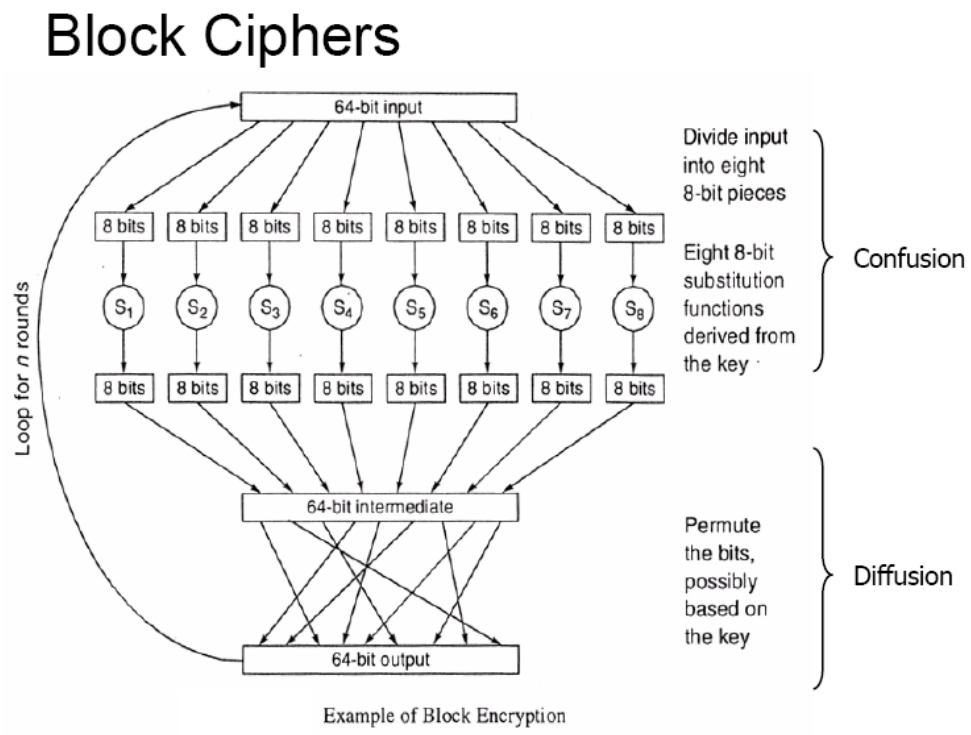


Figura 2.3: Schema a blocchi crittografia a chiave segreta

DES è efficiente se realizzato in hardware, ma relativamente lento se implementato in software. Sebbene l'essere difficilmente implementabile come software non fosse un requisito specificato nel progetto molti sostengono che in realtà questa mancanza fosse voluta, forse per limitarne l'uso ad organizzazioni in grado di realizzare sistemi hardware, o forse perché rese più facile controllare l'accesso alla tecnologia. Ad ogni modo, l'aumento delle capacità di calcolo delle CPU rese possibile realizzare una versione software di DES. Una 500-MIPS (Million Instruction Per Second) CPU può infatti cifrare ad un tasso di circa 30 KB/s e forse più, a seconda dei dettagli architetturali della CPU e dell'intelligenza dell'implementazione. Un processore Intel Core i7 Extreme Edition i980EE ha una capacità di calcolo di circa 150 MIPS, quindi può cifrare ad un tasso di circa 9 KB/s (per cifrare 1 MB impiega circa 110 secondi). L'implementazione software è pertanto attualmente adeguata a molte applicazioni.

Perché chiavi da 56 bit?

La scelta di una chiave da 56 bit causò molte controversie. Prima che DES fu adottato, le persone al di fuori della *intelligence community* lamentavano che 56 bit non offrivano una sicurezza adeguata. Perché solo 56 dei 64 bit di una chiave DES sono effettivamente usati nell'algoritmo? Lo svantaggio di usare 8 bit della chiave per un controllo di parità è che ciò rende DES molto meno sicuro (256 volte meno sicuro contro una ricerca esaustiva). Ma qual è il vantaggio di usare 8 bit per un controllo di parità? Una possibile risposta è che permette di verificare che la chiave non sia corrotta. Tuttavia questa spiegazione non regge. Se si considerassero infatti 64 bit a caso invece della chiave, c'è una probabilità su 256 che il controllo di parità dia esito positivo. La probabilità che la chiave sia comunque errata (nonostante il controllo di parità dia esito positivo) è troppo alta. Inoltre avere una chiave corrotta non comporta un problema di sicurezza, semplicemente la cifratura/decifratura non viene eseguita correttamente. Chiaramente è anche ridicolo sostenere che la scelta di 56 bit sia stata fatta per risparmiare memoria.

La risposta ormai condivisa è che il governo statunitense abbia deliberatamente indebolito la sicurezza di DES di una quantità appena sufficiente da consentire alla NSA di violarlo.

2.3.1 Violabilità e sicurezza di DES

Gli avanzamenti tecnologici dell'industria dei semiconduttori hanno reso ancora più critico il problema della lunghezza della chiave di DES. la velocità dei chip e un po' di furbizia permettono di violare (individuare) le chiavi DES con approcci a forza bruta in tempi ragionevoli. Il rapporto prestazioni/prezzo dell'hardware cresce del 40% per anno. La lunghezza delle chiavi dovrebbe aumentare di 1 bit ogni 2 anni. Assumendo che 56 bit erano appena sufficienti nel 1979 (quando DES fu standardizzato), 64 bit erano adeguati nel 1995, e 128 bit dovrebbero essere sufficienti fino al 2123.

Ragioniamo ora sulla sicurezza di DES. Se si dispone di un singolo blocco $\langle plaintext, ciphertext \rangle$ quanto è difficile trovare la chiave? Un approccio a forza bruta dovrebbe provare ordine di $2^{56} \approx 10^{17}$ chiavi. Se ogni tentativo richiede una singola istruzione sono necessarie ordine di 1000 MIPS-year istruzioni.

- 1 MIPS = 1 Milionef di Istruzioni Per Secondo
- 1 MIPS-year = numero di istruzioni eseguite in un anno ad un tasso pari a 1 MIPS
- 1 MIPS-year = 1 MIPS $\times(365 \times 86400)$ secondi in un anno $= 3,1536 \times 10^{13}$ istruzioni
- $2^{56} \approx 10^{17} \approx 10^3$ MIPS-year

Questo vale tuttavia **SOLO** per effettuare una ricerca esaustiva. Dopo di che entra in gioco il fatto che io sappia riconoscere o meno il testo in chiaro.

Violabilità DES - esempio

Anche nell'ipotesi più scomoda per l'avversario di disporre solo di testo cifrato (in ragionevole quantità), un attacco a forza bruta è ancora possibile. Se ad esempio l'avversario sa soltanto che il testo in chiaro è ASCII a 7 bit ogni volta che prova una chiave deve verificare se sono nulli tutti i bit nelle posizioni $8, 16, 24, ..., n \times 8, ...$ Per ogni blocco di 64 bit, vanno esaminati solo 8 bit; i bit in posizione 8, 16, 24, 32, 40, 48, 56, 64. Se almeno uno di questi bit vale 1 la chiave è sicuramente errata. in caso contrario nulla si può dire; **la probabilità di errore è pertanto 1 su 256 ossia la probabilità che tutti questi bit valgano 0**. Se l'avversario esamina diversi blocchi, ad esempio 10, e verifica che si tratta sempre di ASCII a 7 bit la probabilità che la chiave scelta sia errata si riduce a 1 su 2560. Si noti che gli attuali chip commerciali che implementano DES non si prestano a ricerche esaustive della chiave: sono pensati per cifrare molti dati con una stessa chiave. Infatti il caricamento di una chiave è un'operazione lenta se confrontata con la velocità con la quale viene eseguita la cifratura dei dati. Tuttavia è sempre possibile costruire un chip ottimizzato ad eseguire ricerche esaustive della chiave.

Violabilità DES - cifratura multipla

Per ovviare a tali problemi di sicurezza è possibile cifrare più volte e con diverse chiavi lo stesso blocco di dati. Si parla di **cifratura multipla (multiple encryption)**. Si ritiene che una cifratura con un triplo DES sia 2^{56} volte più difficile da violare.

2.3.2 DES - Struttura base

La struttura di base dell'algoritmo DES è descritta in Figura 2.4. In estrema sintesi:

- L'input di 64 bit è sottoposto ad una permutazione iniziale;
- La chiave da 56 bit viene usata per generare 16 per-round chiavi da 48 bit; una chiave per ciascuno dei 16 round, prendendo 48 differenti sottoinsiemi dei 56 bit della chiave

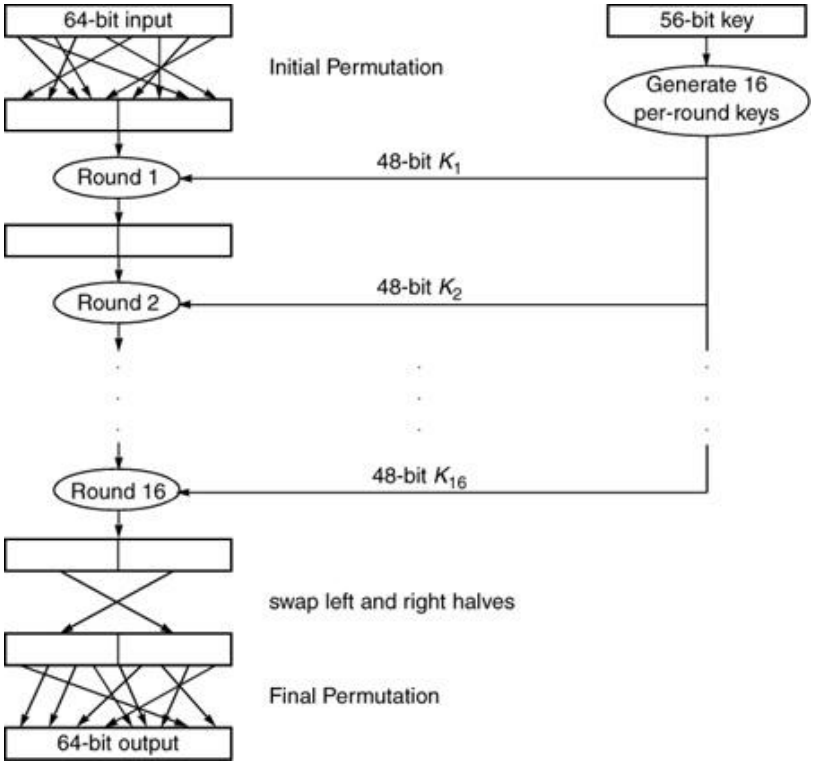


Figura 2.4: Schema a blocchi DES

- Ogni round riceve in input l'output di 64 bit del round precedente e la chiave da 48 bit di quel round
- Ogni round restituisce un output di 64 bit
- Dopo il 16-esimo round, le due metà dell'output di 64 bit vengono scambiate e il risultato viene sottoposto ad un'altra permutazione (inversa a quella iniziale)
- La decifratura consiste semplicemente nell'eseguire la cifratura DES all'indietro
- Per decifrare un blocco è necessario applicare la permutazione iniziale (ciò annulla l'effetto della permutazione finale), generare le 16 chiavi di round, che andranno usate in ordine inverso (prima K_{16}, l'ultima chiave generata), seguire 16 round esattamente come nella cifratura, scambiare le due metà dell'output e sottoporle a un'altra permutazione (che annulla l'effetto della permutazione iniziale)

Per descrivere DES è quindi sufficiente discutere le permutazioni iniziale e finale, come le chiavi di round sono generate, e cosa succede durante un round.

2.3.3 DES - Permutazioni dei dati

L'operazione di permutazione è molto importante in quanto diffonde la dipendenza dei bit in/out. Nel caso di DES però le permutazioni finale e iniziale non hanno effetto. Dimostriamo tale affermazione.

Supponiamo di non averle. Se assumiamo per assurdo che il cifrario cosi ottenuto sia facile da violare, vogliamo dimostrare che anche DES è facile da violare. Chiamiamo il cifrario senza permutazione EDS. Se posso violare EDS (disponendo, ad esempio, di una coppia $\langle plaintext, ciphertext \rangle$), basta permutare ciò che ottengo e violo DES. Infatti, sia $\langle m, c \rangle$ una coppia $\langle plaintext, ciphertext \rangle$ di DES. Si consideri allora la coppia $\langle m', c' \rangle$, dove m' e c' sono ottenuti applicando la permutazione iniziale ad m e c. Allora, la chiave K_{EDS} che si ottiene

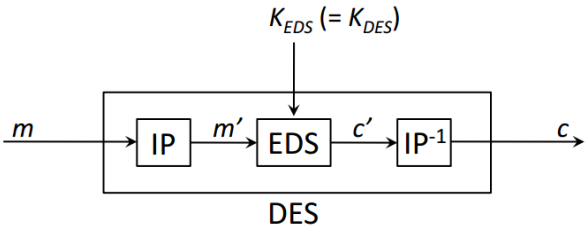


Figura 2.5: Inutilità delle permutazioni iniziale e finale

violando EDS per la coppia $\langle m', c' \rangle$ coincide con la chiave K_{DES} di DES per la coppia $\langle m, c \rangle$ (vedi Figura 2.5). L'ipotesi è che tali permutazioni siano state introdotte per rendere più complicata una possibile implementazione software. Come sono state definite quindi le permutazioni

Initial Permutation (IP)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Final Permutation (IP^{-1})

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Figura 2.6: Tabella permutazioni DES

iniziale e finale? E' di fatto una permutazione regolare, come è mostrato in Figura 2.6. Le tabelle vanno interpretate nel seguente modo: i numeri, da 1 a 64, riportati in tabella rappresentano le posizioni dei bit in input alla permutazione, mentre l'ordine (per righe da sx verso dx) dei numeri nella tabella rappresenta la corrispondente posizione dei bit in output. Ad esempio, la permutazione iniziale sposta il 58-esimo bit in input nel primo bit in output, e il 50-esimo bit in input nel secondo bit in output. Non si tratta di permutazioni generate in modo random, in quanto presenta delle evidenti regolarità, come è evidenziato in Figura 2.7.

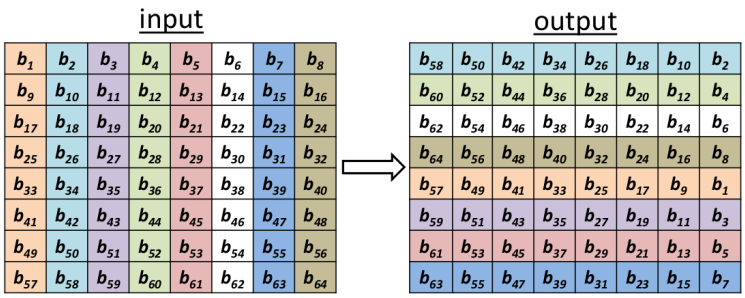


Figura 2.7: Regolarità tabella permutazioni DES

2.3.4 DES - Generazione chiavi di round

DES genera sedici chiavi di round da 48 bit a partire dalla chiave principale K di 64 bit nominali, di cui solo 56 effettivi. I bit di parità di K non vengono infatti considerati. Denoteremo con K_1,K_2,...,K_{16} le sedici chiavi di round. Il procedimento è il seguente:

- viene prima effettuata una permutazione iniziale sui 56 bit effettivi di K
- i 56 bit in output vengono divisi in due metà C_0 e D_0
- Le sedici chiavi vengono generate in sedici round, i.e. nell'i-esimo round viene generata la chiave di round K_i

Permutazione iniziale

C_0

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36

D_0

63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Figura 2.8: Permutazione iniziale della chiave

input

b_1	b_2	b_3	b_4	b_5	b_6	b_7
b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
b_{17}	b_{18}	b_{19}	b_{20}	b_{21}	b_{22}	b_{23}
b_{25}	b_{26}	b_{27}	b_{28}	b_{29}	b_{30}	b_{31}
b_{33}	b_{34}	b_{35}	b_{36}	b_{37}	b_{38}	b_{39}
b_{41}	b_{42}	b_{43}	b_{44}	b_{45}	b_{46}	b_{47}
b_{49}	b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}
b_{57}	b_{58}	b_{59}	b_{60}	b_{61}	b_{62}	b_{63}

output

b_{57}	b_{49}	b_{41}	b_{33}	b_{25}	b_{17}	b_9
b_1	b_{58}	b_{50}	b_{42}	b_{34}	b_{26}	b_{18}
b_{10}	b_2	b_{59}	b_{51}	b_{43}	b_{35}	b_{27}
b_{19}	b_{11}	b_3	b_{60}	b_{52}	b_{44}	b_{36}
b_{63}	b_{55}	b_{47}	b_{39}	b_{31}	b_{23}	b_{15}
b_7	b_{62}	b_{54}	b_{46}	b_{38}	b_{30}	b_{22}
b_{14}	b_6	b_{61}	b_{53}	b_{45}	b_{37}	b_{29}
b_{21}	b_{13}	b_5	b_{28}	b_{20}	b_{12}	b_4

Figura 2.9: Regolarità tabella permutazioni delle chiavi DES

La struttura della permutazione iniziale della chiave è illustrata in Figura 2.6. Il valore numerico di un elemento della tabella rappresenta la posizione del bit in input, mentre l'ordine nella tabella rappresenta la posizione del bit in output (come per le permutazioni dei dati). Anche in questo caso non è random, in quanto presenta delle evidenti regolarità, come è evidenziato in Figura 2.9. Come nel caso delle permutazioni iniziale e finale dei dati, non apportano alcun miglioramento alla sicurezza.

Generazione K_i nell'i-esimo round

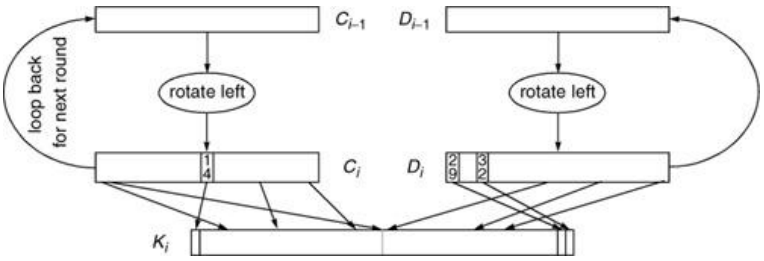


Figura 2.10: generazione di Ki: round i

Trattiamo ora la generazione della chiave K_i che avviene nell'i-esimo round. La procedura avviene seguendo questi passi (come mostrato in Figura 2.10):

- i bit delle due metà C_{i-1} e D_{i-1} della (i-1)-esima chiave K_{i-1} vengono ruotati (traslazione ciclica) a sinistra. L'entità della traslazione a dipende dal round: nei round 1, 2, 9 e 16 si ha una rotazione a sinistra di un solo bit (i.e. il primo bit diventa l'ultimo bit a destra), mentre negli altri round si ha una rotazione a sinistra di due bit.
- la permutazione di C_i che produce la metà sinistra di K_i è illustrata sotto. Si noti che i bit in posizione 9, 18, 22, e 25 sono scartati.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2

- la permutazione di D_{i-1} ruotato, cioè di D_i , che produce la metà destra di K_i è illustrata sotto. Si noti che i bit in posizione 35, 38, 43, e 54 sono scartati; rimangono così 24 bit anziché 28.

41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

2.3.5 Un round di DES

La struttura di un round di des è illustrata in Figura 2.11). I 64 bit in input sono divisi in due metà da 32 bit: L_n, la metà sinistra dopo l'(n-1)-esimo round, e R_n, la metà destra dopo l'(n-1)-esimo round. L'output di 64 bit del round si ottiene concatenando le due metà: L_{n+1}, la metà sinistra dopo l'n-esimo round, e R_{n+1}, la metà destra dopo l'n-esimo round. L_{n+1} è semplicemente R_n, mentre R_{n+1} è ottenuto come segue:

- R_n e K_n sono posti in input alla mangler function; la mangler function viene anche detta funzione di Feistel
- l'output della mangler function, $mangler(R_n, K_n)$, è una quantità di 32 bit; si noti che R_n, K_n sono composti da 32 e 48 bit rispettivamente
- l'output $mangler(R_n, K_n)$ viene poi sommato (XOR) con L_n
- il risultato ottenuto è $R_{n+1} = mangler(R_n, K_n) \oplus L_n$

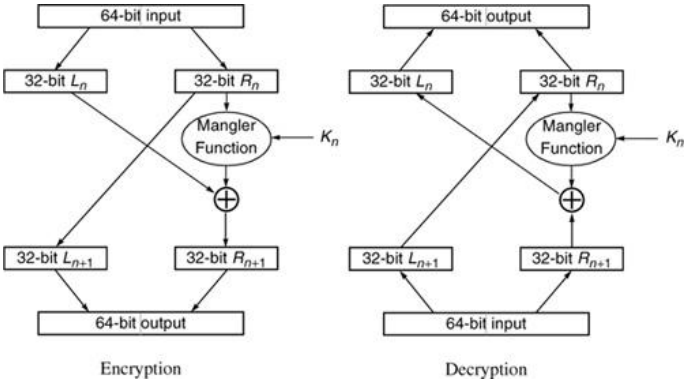


Figura 2.11: Struttura di un round di DES

Si noti che ogni round di DES è facilmente invertibile: noti L_{n+1} , R_{n+1} e k_n è facile ottenere L_n e R_n. Infatti $R_n = L_{n+1}$ e, poiché $R_{n+1} = mangler(R_n, K_n) \oplus L_n$, risulta $R_{n+1} \oplus mangler(R_n, K_n) = L_n$ (in virtù della proprietà dello XOR secondo cui $x \oplus y \oplus y = x$). La mangler function non è quindi mai utilizzata in senso inverso. DES è elegantemento progettato in modo da essere facilmente invertibile senza richiedere l'invertibilità della mangler function. Esaminando attentamente la Figura 2.11), si evince che la decifratura è di fatto identica alla cifratura, tranne per il fatto che le due metà da 32 bit sono invertite. In altre parole, fornendo $R_{n+1} \mid L_{n+1}$ in input al round n si ottiene $R_n \mid L_n$ in uscita

2.3.6 Mangler Function

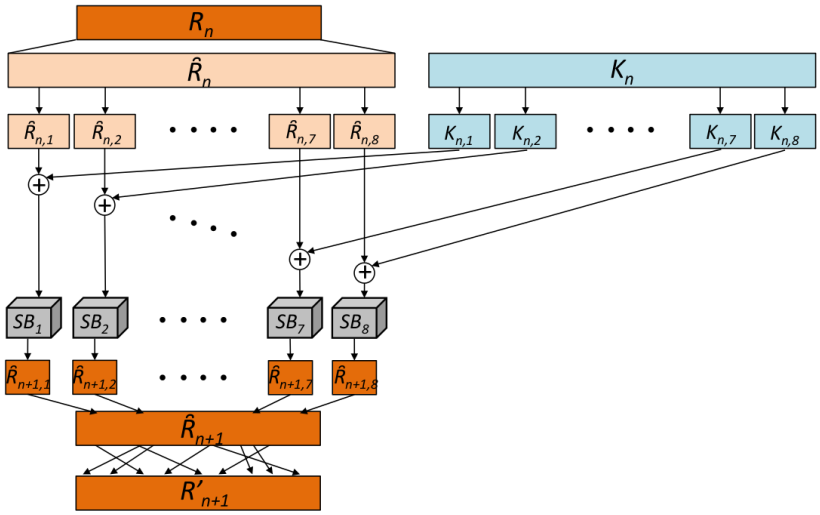


Figura 2.12: Mangler function

La mangler function prende in input i 32 bit di R_n, o R per semplificare, e i 48 bit della chiave K_n , o K per semplificare. Produce un output da 32 bit che sommato (XOR) con L_n permette di ottenere R_{n+1} (il prossimo R).
La prima operazione è l'espansione di R, da 32 bit a 48 bit. R è scomposto in otto pezzi da 4 bit $R = \{r_1, r_2, ..., r_8\}$. L'i-esimo pezzo r_i viene espanso a 6 bit aggiungendo in testa e in coda rispettivamente l'ultimo bit di r_{i-1} e il primo bit di r_{i+1}. r_1 e r_8 sono considerati adiacenti, cioè $r_0 = r_8$ e $r_9 = r_1$. (Fare riferimento alla Figura 2.13)).
la chiave K(K_n) viene scomposta in otto chunk (pezzi) da 6 bit. L'i-esimo chunk di R(R_n) espanso viene sommato (XOR) all'i-esimo chunk di K. L'output a 6 bit ottenuto viene sottoposto

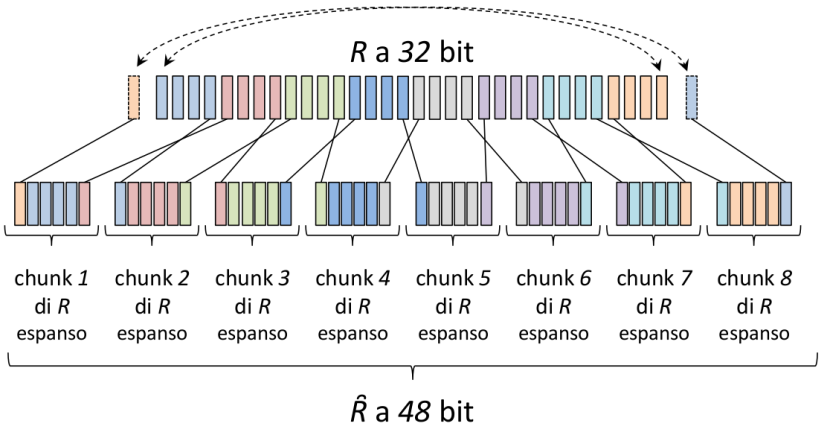


Figura 2.13: Mangler function - espansione

ad una sostituzione **S-box** che produce un output di 4 bit. Vi sono in tutto otto S-box distinte; l'i-esima S-box elabora la somma dell'i-esimo chunk di K e di R. Ogni S-box ha 64 possibili input e 16 possibili output. Ovviamente input diversi possono essere mappati nello stesso output (sto riducendo lo spazio). Le S-box sono definite in modo tale che esattamente quattro input distinti sono mappati in ciascun output possibile. Ogni S-box può riguardarsi come quattro S-box separate aventi 4 bit sia in input che in output: i quattro bit in input corrispondono ai bit interni (dal secondo al quinto) dell'input globale, e i due bit esterni (primo e sesto) dell'input globale servono a selezionare quale dei quattro output ottenuti rappresenta l'output globale. Infine gli

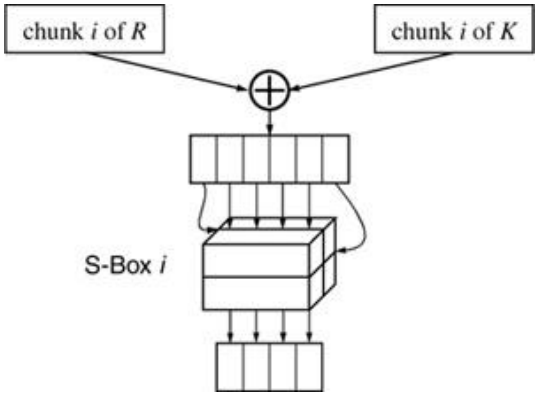


Figura 2.14: S-Box

otto output, da 4 bit, delle otto S-box sono riuniti in un unico output a 32 bit che viene sottoposto ad una permutazione. Tale permutazione aumenta il livello di sicurezza perché le sostituzioni fatte in ciascuna S-box in un round di DES vengono diffuse negli input di più S-box nel round seguente. Senza tale permutazione, un bit nella parte sinistra dell'input influenzerebbe principalmente alcuni bit della parte sinistra dell'output. Tale permutazione è mostrata in Figura 2.15).

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10	2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Figura 2.15: Permutazione output S-Box

2.4 Advanced Encryption Standard - AES

AES fu introdotto perché DES aveva una chiave troppo corta, 3DES (triplo DES) era troppo lento e IDEA era protetto da un brevetto, ed era in parte sospetto e lento. Il NIST si impegnò quindi a sviluppare un nuovo standard. Non si trattava di un problema solo tecnico, ma anche di un problema politico, poiché alcuni rami del governo avevano ostacolato il più possibile la diffusione e l'esportazione della crittografia sicura, e il fatto che il governo appoggiasse il NIST era visto con scetticismo, e molti non si fidavano. Il NIST voleva realmente creare un nuovo standard di sicurezza eccellente, cioè efficiente, flessibile, sicuro e free (non protetto).

Nel 1997 il NIST annunciò una gara per la selezione di un nuovo standard di cifratura destinato a proteggere informazioni governative sensibili. Dopo molti anni di studio e discussioni, il NIST scelse l'algoritmo **Rijndael**, proposto da due crittografi belgi Joan Daemen e Vincet Rijmen. Rijndael prevede diverse lunghezze per i blocchi e per la chiave: 128, 160, 192, 224, e 256 bit. La lunghezza di un blocco e quella della chiave possono differire. Il 26 Novembre 2001, viene emanato **AES**, una standardizzazione di Rijndael. AES quindi consiste in un'implementazione di **Rijndael** con determinati parametri. Lo standard AES, infatti:

- fissa la lunghezza dei blocchi a 128 bit
- la chiave può essere di 128, di 192 o di 256 bit. Si parla di **AES-128**, **AES-192** e **AES-256**

Nel seguito si descriverà Rijndael, specificando di volta in volta i parametri di AES. A grandi linee, Rijndael somiglia a DES e a IDEA. Ci sono più round che "strapazzano" un blocco di testo in chiaro per ottenere il corrispondente cifrato, e c'è un algoritmo per l'espansione della chiave; a partire dalla chiave segreta genera le chiavi da usare nei vari round.

Rijndael/AES - parametri

Rijndael ha una struttura flessibile grazie all'uso di due parametri indipendenti, e di un terzo parametro derivato dai primi due:

- N_b **dimensione di un blocco**: numero di parole (word) da 32 bit (colonne da 4 ottetti) in un blocco da cifrare. In AES $N_{b=4}$, ovvero un blocco ha lunghezza 128 bit, ovvero 4 parole da 32 bit (4 colonne da 4 ottetti).
- N_k **dimensione della chiave**: numero di parole (word) da 32 bit (colonne da 4 ottetti) in una chiave di cifratura. In AES-128 $N_k = 4$, In AES-192 $N_k = 6$, In AES-256 $N_k = 8$. In Rijndael N_k può essere un qualsiasi intero tra 4 e 8.
- N_r **numero di round**: questo parametro dipende da N_b e da N_k. Il numero di round deve aumentare all'aumentare della lunghezza di un blocco (e della chiave); ogni bit del testo in chiaro (e della chiave) deve influenzare (in modo complesso) ciascun bit del testo cifrato. Rijndael specifica che $N_r = 6 + \max(N_b, N_k)$. In AES-128 $N_r = 10$, in AES-192 $N_r = 12$, in AES-256 $N_r = 14$.

Array di stato

Rijndael mantiene un array di stato rettangolare durante il funzionamento. Ogni elemento dell'array è un ottetto. Complessivamente ci sono N_b colonne da 4 ottetti. Lo stato iniziale è ottenuto popolando l'array, colonna per colonna, mediante le N_b colonne da 4 ottetti che costituiscono il blocco di input.

Capitolo 3

Modes of Operation

3.1 Introduzione

Si illustrerà come usare gli algoritmi di crittografia a chiave segreta, DES, IDEA e AES, in applicazioni reali: è stato visto soltanto come usare tali algoritmi per cifrare blocchi di lunghezza prefissata (64 bit per DES e IDEA, 128 bit per AES), come si procede se è necessario cifrare dei messaggi di lunghezza arbitraria/diversa?

Si vedrà inoltre come si generano dei MAC (**M**essage **A**uthentication **C**ode) sfruttando la crittografia a chiave segreta.

3.2 Cifrare messaggi di grandi dimensioni

Come è possibile cifrare messaggi di dimensioni superiori a 64 bit?
Sono state proposte diverse **modalità operative dei cifrari a blocchi**, cioè ci si è interrogati su come utilizzare i cifrari a blocchi nel caso di messaggi di lunghezza maggiore di quella di un singolo blocco.

Le modalità più conosciute e che verranno descritte di seguito sono:

- **E**lectronic **C**ode **B**ook (ECB)
- **C**ipher **B**lock **C**haining (CBC)
- k-Bit **C**ipher **F**eed**B**ack Mode (CFB)
- k-Bit **O**utput **F**eed**B**ack Mode (OFB)
- **C**oun**T**e**R** Mode (CTR)

Si noti che nel seguito si farà riferimento a cifrari a blocchi con blocchi di 64 bit (tutte le considerazioni valgono anche nel casi di cifrari con blocchi di dimensione diversa da 64 bit).

3.2.1 Electronic Code Book (ECB)

Questa modalità consiste nel fare la cosa più ovvia, ma corrisponde, in genere, alla soluzione peggiore, cioè:

- il messaggio viene decomposto in blocchi da 64 bit (inserendo eventualmente dei bit di padding nell'ultimo blocco al fine di riempirlo)
- ciascun blocco da 64 bit viene cifrato con la chiave segreta
- ciascun blocco cifrato viene decifrato

- il messaggio viene ricomposto a partire dai singoli blocchi decifrati

Come illustrato nella figure Figura 3.1 e Figura 3.2

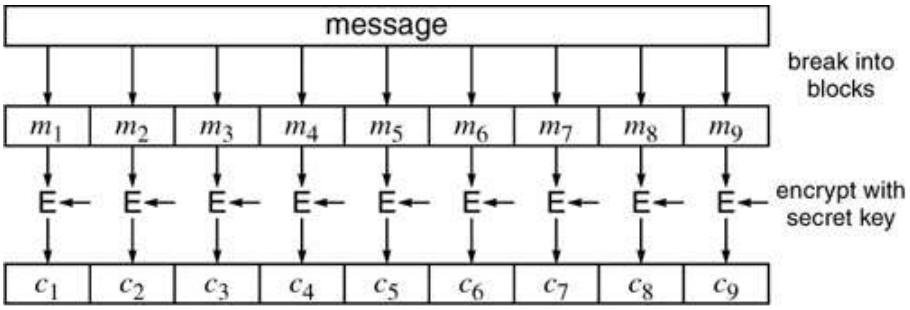


Figura 3.1: Schema di crittografia ECB

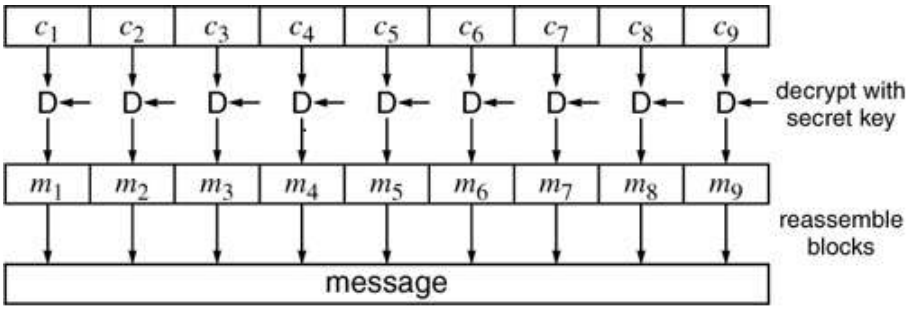


Figura 3.2: Schema di decrittografia ECB

Problemi di sicurezza in ECB

La modalità operativa ECB introduce una serie di problemi non presenti nel cifrario a blocchi: se il messaggio contiene due blocchi di 64 bit identici, allora anche i corrispondenti blocchi cifrati saranno identici e ciò fornisce delle informazioni aggiuntive sul testo in chiaro che un ascoltatore può sfruttare. Si consideri ad esempio lo scenario della Figura 3.3: supponiamo che l'ascoltatore sappia che il testo in chiaro contiene l'elenco, ordinato alfabeticamente, degli impiegati e dei relativi salari inviato dall'amministrazione all'ufficio paghe, supponiamo inoltre che ogni riga del file sia lunga esattamente 64 byte (8 blocchi da 8 byte) e che i vari blocchi risultino suddivisi in modo tale che alcuni contengono la codifica della cifra decimale più significativa del campo salario (migliaia di dollari/euro). Comparando i testi cifrati, l'ascoltatore, oltre a dedurre quanti

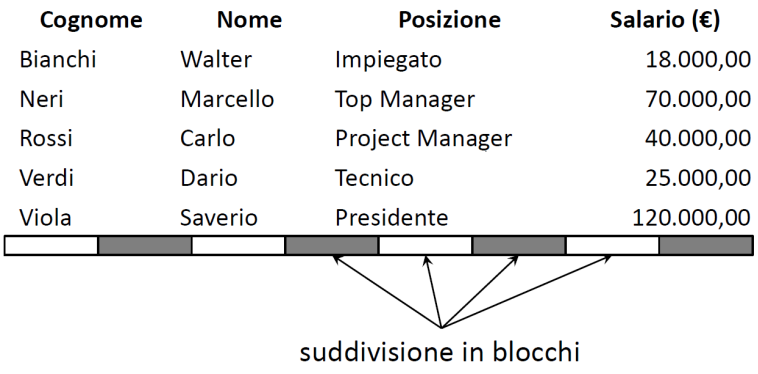


Figura 3.3: File con i salari, oggetto di attacco

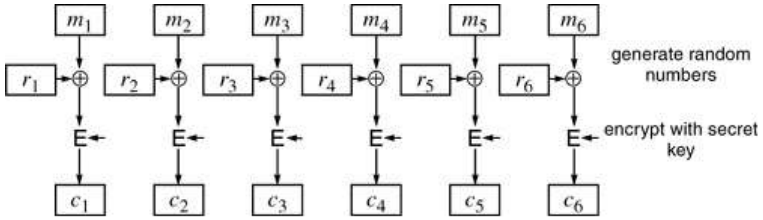


Figura 3.4: Randomized Electronic Code Book Encryption

dipendenti hanno lo stesso salario, può anche dedurre quanti dipendenti hanno uno stipendio nello stesso range (ordine di 10 euro/dollari); se ci sono complessivamente pochi range salariali, l'ascoltatore può dedurre a quale categoria di dipendente corrisponda un dato blocco cifrato; inoltre, se l'ascoltatore è un impiegato, può sostituire il blocco cifrato di un altro dipendente (un manager) al suo blocco cifrato (dedotto in base all'ordine e alla numerosità della sua classe salariale).

ECB quindi ha due serie debolezze, legate al fatto che qualcuno, analizzando diversi blocchi cifrati potrebbe:

- dedurre (inferire) informazioni sfruttando le ripetizioni di alcuni blocchi
- riarrangiare/modificare i blocchi cifrati a proprio vantaggio

Per tali ragioni ECB è raramente usato.

3.2.2 Cipher Block Chaining (CBC)

CBC non presenta i problemi di ECB: a due blocchi in chiaro identici non corrispondono due blocchi cifrati identici.

Idea base

Per comprendere CBC conviene prima considerare il seguente esempio, in riferimento alla Figura 3.4 che ne condivide l'idea base:

- per ogni blocco di testo in chiaro mi viene generato un numero random a 64 bit r_i
- m_i e r_i vengono sommati ($\oplus$ XOR)
- il risultato viene cifrato con la chiave segreta
- i blocchi cifrati c_i e i numeri random, in chiaro, r_i vengono trasmessi

E per riottenere il testo in chiaro:

- vengono decifrati i blocchi c_i con la chiave segreta
- i blocchi risultanti vengono sommati ($\oplus$ XOR) con i numeri random r_i

L'esempio appena visto è molto inefficiente, infatti l'informazione da trasmettere è duplicata perché per ogni blocco va trasmesso il corrispondente numero random.
Un altro problema è che un avversario può riarrangiare i blocchi in modo da ottenere un effetto predittivo sul testo in chiaro, ad esempio:

- se la coppia $r_2|c_2$ fosse rimossa il corrispondente blocco in chiaro m_2 scomparirebbe
- se la coppia $r_2|c_2$ fosse scambiata con la coppia $r_7|c_7 \Rightarrow m_2$ e m_7 risulterebbero scambiati
- se l'avversario conosce ciascun m_i, può modificare m_i in modo predittivo cambiando il corrispondente numero random r_i

Funzionamento

CBC genera i "propri" numeri random usando c_i come numero random r_{i+1}, cioè usa il precedente blocco cifrato come numero random da sommare ($\oplus$ XOR) al blocco di testo in chiaro successivo.
Per evitare che due testi in chiaro inizialmente identici diano luogo a dei blocchi cifrati inizialmente identici CBC genera un singolo numero random, detto vettore di inizializzazione (**Initialization Vector** IV), che viene sommato ($\oplus$ XOR) con il primo blocco di testo in chiaro. Il risultato viene trasmesso dopo la cifratura a chiave segreta.
La decifratura è semplice essendo l'or-esclusivo un'operazione che coincide con la propria inversa.
Quanto detto, rappresentato nelle figure Figura 3.5 e Figura 3.6, può essere espresso anche algebricamente:

- CIFRATURA
  $c_1 = E(K, (IV \oplus m_1))$
  $c_i = E(K, (c_{i-1} \oplus m_i)) \forall i > 1$
- DECIFRATURA
  $m_1 = E(K, c_1) \oplus IV$
  $m_i = E(K, c_i) \oplus c_{i-1}$

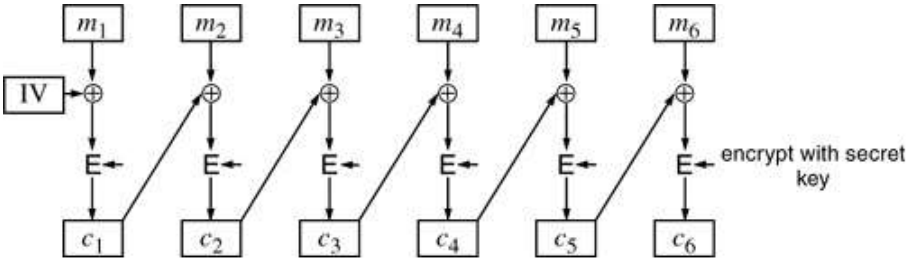


Figura 3.5: Schema di crittografia CBC

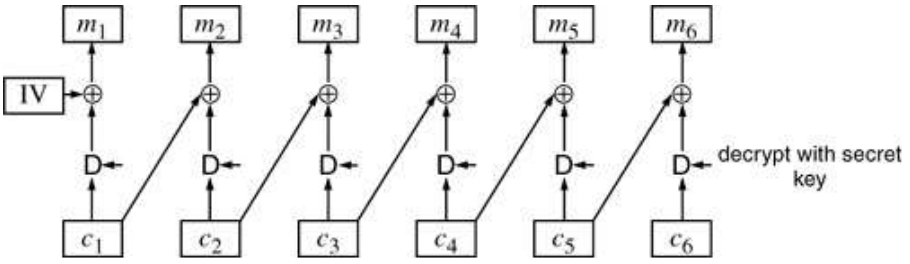


Figura 3.6: Schema di decrittografia CBC

Si noti che, essendo il costo della somma ($\oplus$ XOR) trascurabile rispetto al costo della cifratura a chiave segreta, la cifratura con CBC ha le stesse prestazioni della cifratura con ECB eccetto il costo delle generazione e trasmissione di IV. In molti casi, tuttavia, la sicurezza di CBC non dipende dalla scelta del vettore di inizializzazione IV (cioè si possono anche porre tutte le cifre di IV pari a 0).

In alcuni casi, tuttavia, l'assenza di IV riduce la sicurezza. Ad esempio, si supponga che il file cifrato contenente i salari dei dipendenti sia trasmesso settimanalmente; in assenza di IV, un ascoltatore potrebbe verificare se il testo cifrato differisce da quelle della precedente settimana, e potrebbe determinare la prima persona il cui salario è cambiato. Un altro esempio è quello di un generale che invia giornalmente delle informazioni segrete dicendo "continue holding your position"; il testo cifrato sarebbe ogni giorno lo stesso, finché il generale decide di cambiare ordine, inviando il messaggio "start bombing": il testo cifrato cambierebbe immediatamente, allertando

il nemico.
Un vettore di inizializzazione scelto randomicamente garantisce che, anche se lo stesso messaggio è inviato ripetutamente, il corrispondente testo cifrato risulta ogni volta differente, e previene attacchi all'algoritmo di cifratura di tipo testo in chiaro selezionato anche quando un avversario può fornire del testo in chiaro al CBC.

CBC minaccia 1 – Modifica dei blocchi cifrati

L'uso di CBC non elimina il problema che qualcuno possa modificare il messaggio in transito, semplicemente cambia la natura della minaccia: un avversario non può più vedere ripetizioni di blocchi cifrati, e non può più copiare/spostare blocchi cifrati (ad esempio per scambiare il salario di due dipendenti) ma può ancora modificare il testo cifrato in modo predittivo.
Cosa potrebbe succedere se modificasse un blocco di testo cifrato, ad esempio c_n?
Da $m_{n+1} = E(K, c_{n+1}) \oplus c_n$ si evince che una modifica di c_n può avere un effetto prevedibile su m_{n+1} (ad esempio, cambiando il terzo bit di c_n cambia il terzo bit di m_{n+1}); chiaramente essendo anche $m_n = E(K, c_n) \oplus c_{n-1}$, l'avversario non può prevedere quale possa essere il nuovo valore di m_n, molto probabilmente la modifica di c_n corrompe completamente il blocco in chiaro m_n.
Vediamo a tal proposito l'esempio seguente, illustrato in Figura 3.7:

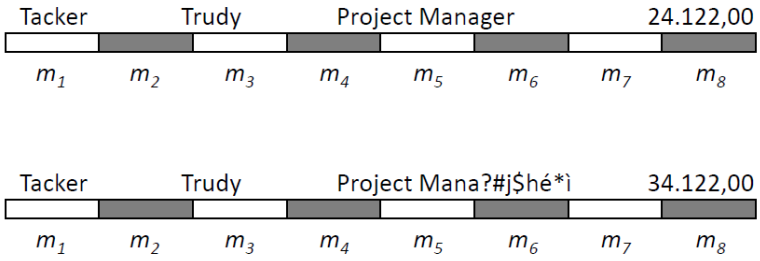


Figura 3.7: Modifica dei blocchi cifrati

Supponiamo che un avversario (Trudy) sappia che una data sequenza di blocchi cifrati, del file dei salari, corrispondano alla riga contenente i suoi dati personali; se Trudy vuole aumentare il suo salario di 10K, e se sa che l'ultimo byte di m_7 corrisponde alle decine di migliaia nella codifica decimale (00000010), per darsi 10K in più deve semplicemente cambiare il bit meno significativo di c_6; da $m_7 = D(K, c_7) \oplus c_6$ tuttavia, Trudy non sarà più in grado di predire cosa apparirà nella voce "Posizione", infatti, da $m_6 = D(K, c_6) \oplus c_5$, si vede che è impraticabile prevedere l'effetto della modifica di c_6 su m_6. Se il file decifrato fosse letto da una persona umana, questa potrebbe insospettirsi della presenza di simboli strani nel campo "Posizione", se invece il file decifrato viene elaborato da un programma l'attacco potrebbe non essere rilevato.

Ricapitolando: Trudy è stato in grado di modificare un blocco in modo predittivo con l'effetto collaterale di modificare il blocco precedente senza poter prevedere il risultato finale.

CBC minaccia 2 – Riarrangiamento blocchi cifrati

Con riferimento alla Figura 3.8, si supponga che Trudy conosca il testo in chiaro e, il corrispondente testo cifrato di qualche messaggio, cioè m_1, m_2, ..., m_n e IV, c_1, c_2, ..., c_n; in questo modo Trudy conosce automaticamente anche il blocco decifrato di ciascun c_i, da $D(K, c_1) = c_{i-1} \oplus m_i$. Da queste informazioni, Trudy può considerare ciascun c_i come un "building block" e costruire un flusso cifrato usando ogni combinazione di c_i ed essere in grado di calcolare quale sarà il corrispondente testo in chiaro.

Per capire a cosa potrebbe servire questo tipo di attacco si tenga presente che uno dei modi di combattere la minaccia di modifica di blocchi cifrati è includere un **CRC** (**C**yclic **R**edundancy

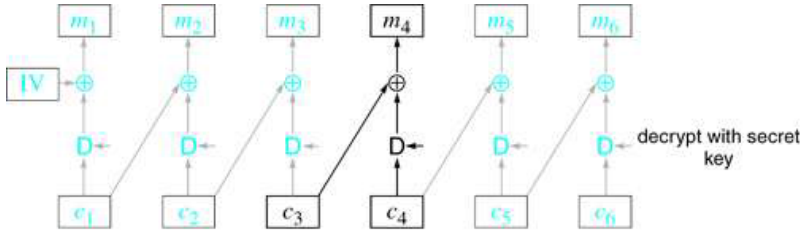


Figura 3.8: Riarrangiamento dei blocchi cifrati

Check) al testo in chiaro prima di cifrarlo con un CBC, perciò se Trudy modifica qualche blocco cifrato, il CRC consentirà ad un computer di rilevare prontamente l'alterazione del messaggio (e.g. se si fosse scelto un CRC a 32 bit ci sarebbe una possibilità su 2^32 che il CRC coincida con quello corretto); supponiamo che a Trudy non interessi quale possa essere il nuovo messaggio di testo in chiaro (che potrebbe essere completamente indecifrabile), ma interessi solamente che il nuovo messaggio manomesso sia accettato dal computer ricevente sapendo che viene eseguito un controllo di tipo CRC, Trudy può provare a costruire molti flussi cifrati combinando in modi diversi i blocchi c_1, c_2, ..., c_n e può calcolare il risultante testo in chiaro per ciascuno di essi, per poi testare se il testo in chiaro risultante ha un CRC corretto (mediamente serviranno 2^31 tentativi).
Che male potrebbe fare Trudy modificando un messaggio, senza controllarne il contenuto, in modo tale che sia accettato dal computer ricevente? Forse Trudy è soltanto maliziosa, e vuole distruggere alcuni dati che vengono caricati attraverso la rete, ma in realtà, c'è un modo sottile di controllare, seppur in misura ridotta, il contenuto del messaggio modificato: supponiamo che Trudy sposti blocchi contigui, ad esempio, se c_n e c_{n+1} vengono spostati in qualche altro posto, allora il blocco originale m_{n+1} apparirà in un'altra posizione; se m_{n+1} contiene il salario del presidente, Trudy potrebbe scambiare i blocchi in modo da cambiarlo con il suo, ma poi dovrà modificare molto probabilmente gran parte del messaggio restante per garantire che il CRC risulti invariato.

Per prevenire attacchi di questo tipo, basati sul riarrangiamento dei blocchi cifrati e tali da preservare il CRC originario, potrebbe essere usato un CRC a 64 bit: ciò è sicuramente sufficiente se l'attacco al CRC, nell'ambito di un CBC, è di tipo a forza bruta.
Per chi progetta protocolli crittografici sicuri, una modalità di cifratura ad un singolo step che protegga sia la confidenzialità che l'autenticità di un messaggio è stata per molti anni una sorta di "Sacro Graal" da ricercare!

3.2.3 Output FeedBack Mode (OFB)

L'OFB è un cifrario a flusso: la cifratura consiste nel sommare ($\oplus$ XOR) il messaggio con il keystream (o one-time pad) generato da OFB stesso. Supponiamo che il keystream sia ottenuto generando singoli blocchi di 64 bit alla volta; un possibile modo per generarlo è il seguente:

- viene generato un numero random IV (Initialization Vector) di 64 bit
- il primo blocco del keystream coincide con IV: $b_0 = IV$
- i blocchi seguenti b_i si ottengono cifrando b_{i-1} con la chiave segreta: $b_i = E(K, b_{i-1})$

Il one-time pad (keystream) risultante è dato dalla sequenza $OTP = OTP(K, IV) = b_0|b_1|b_2|\ldots|b_i|b_{i+1}|\ldots$.
La cifratura con OFB (Figura 3.9) consiste nel sommare ($\oplus$ XOR) il messaggio m con OTP, se m ha lunghezza l_m bit si considereranno soltanto l_m bit di OTP. Il risultato della cifratura $c = OTP \oplus m$ viene trasmesso insieme a IV (la lunghezza l_c di c coincide con l_m).
In decifratura (Figura 3.10) il destinatario riceve IV e conoscendo K calcola lo stesso onetime pad $OTP = OTP(K, IV)$ il messaggio m si ottiene sommando ($\oplus$ XOR) il flusso cifrato c con l_c bit di OTP.

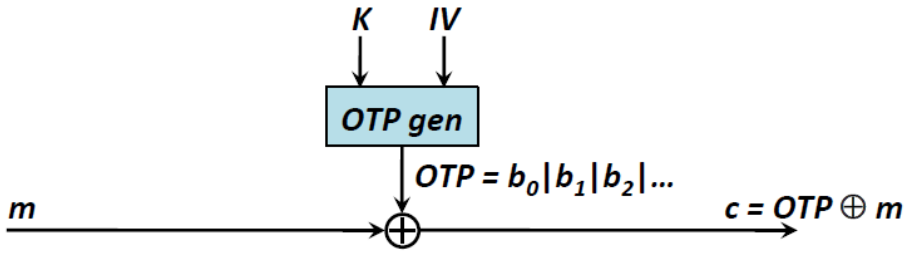


Figura 3.9: Schema di crittografia OFB

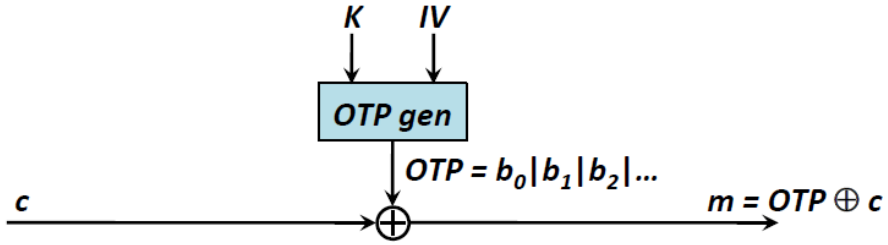


Figura 3.10: Schema di decrittografia OFB

Vantaggi e svantaggi di OFB

OFB presenta i seguenti vantaggi:

- visto che il one-time pad OTP può essere generato in anticipo, prima che sia noto il messaggio m da cifrare, una volta ottenuto m è necessario soltanto effettuare la somma ($\oplus$ XOR) con il one-time pad (lo XOR è eseguibile in modo estremamente veloce)
- se qualche bit del testo cifrato dovesse corrompersi, soltanto i corrispondenti bit del testo in chiaro sarebbero corrotti, diversamente dalla modalità CBC, dove se c_n fosse corrotto allora m_n sarebbe completamente corrotto e m_{n+1} sarebbe corrotto in corrispondenza dei medesimi bit di c_n
- un messaggio m può arrivare a pezzi di lunghezza arbitraria, e, ogni volta che arriva un pezzo, il corrispondente testo cifrato può essere immediatamente trasmesso; in CBC invece, se il messaggio arriva un byte alla volta, per la cifratura è comunque necessario attendere che un blocco di 64 bit (o un multiplo intero di 8 byte) sia completo (ciò può comportare l'attesa di altri 7 byte o l'aggiunta di bit di riempimento, cosa che aumenta la quantità di dati da trasmettere)

OTP ha però anche il seguente svantaggio:

- se un avversario conoscesse il testo in chiaro m e quello cifrato c, potrebbe modificare il testo in chiaro a piacimento semplicemente sommando il testo cifrato con il testo in chiaro noto, e sommando il risultato con un qualsiasi messaggio m' che desidera sostituire ad m; cioè l'avversario dovrebbe modificare il testo cifrato come $c' = c \oplus m \oplus m'$ e verificare che decifrando c' anziché c si ottiene m' anziché m.

3.2.4 k-bit Output FeedBack Mode (k-OFB)

In generale la modalità OFB consente di generare flussi di "pezzi" da k bit (quanto visto prima corrisponde al caso in cui $k = 64$ bit). La modalità k-bit OFB funziona nel seguente modo (si descriverà la versione data in [DES81]) con riferimento alla Figura 3.11:

- l'input I_0 al modulo di cifratura DES è inizializzato a IV, cioè $I_0 = IV$: se IV ha meno di 64 bit, vengono inseriti degli 0 di riempimento a sinistra (cifre più significative)

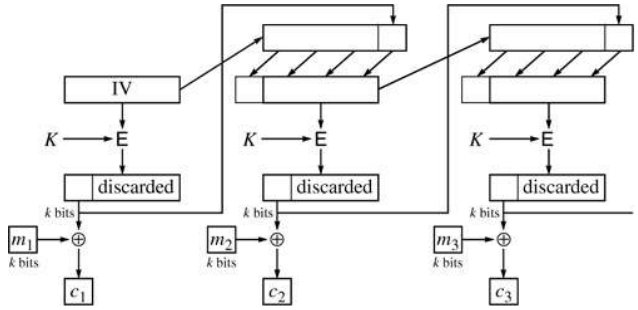


Figura 3.11: k-bit OFB

- il primo pezzo b_0 di OTP si ottiene selezionando k bit dall'output $O_0 = E(K, I_0)$ di DES (una quantità a 64 bit); da un punto di vista crittografico non ha importanza come siano scelti tali bit da O_0; [DES81] specifica che devono essere i k bit più significativi
- l'i-esimo pezzo b_i si ottiene selezionando i k bit più significativi dell'output $O_i = E(K, I_i)$ di DES, ove l'input I_i è stato ottenuto da I_{i-1} eseguendo una traslazione a sinistra di k bit, e un inserimento di b_{i-1} nei k bit meno significativi di I_1 (k bit più a destra)

3.2.5 Cipher FeedBack Mode (CFB)

La modalità CFB è molto simile a OFB, con riferimento alla Figura 3.12:

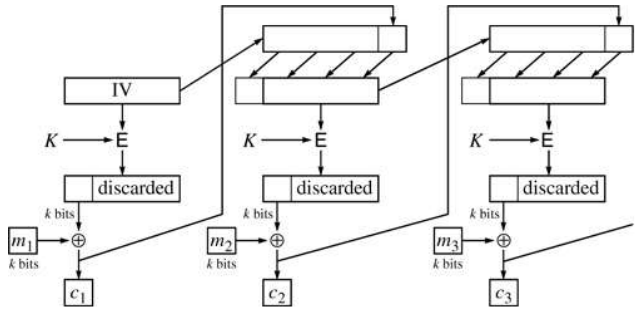


Figura 3.12: k-bit CFB

- viene prodotto un one-time pad generando, uno alla volta, singoli pezzi di k bit
- il one-time pad viene sommato ($\oplus$ XOR) con pezzi di k bit del messaggio

Si noti che in OFB i k bit meno significativi dell'input I_i del modulo di cifratura DES sono i k bit di b_{i-1} (sono parte dell'output O_{i-1} della cifratura DES del blocco precedente; invece, in CFB i k bit di I_i sono i k bit di testo cifrato del blocco precedente, cioè i k bit di c_{i-1} (in CFB il one-time pad non può essere generato prima che il messaggio e noto, a differenza di OFB), nella modalità a k-bit è ragionevole assegnare a k un valore diverso da 64 bit (una scelta sensata è k = 8 bit).

Vantaggi e svantaggi di CFB

CBF presenta i seguenti vantaggi:

- Con OFB o CBC, se si ha una perdita di caratteri in trasmissione (testo cifrato), i.e. se nel flusso cifrato $c_1, c_2, c_3, ..., c_n, ...$ si perde il carattere c_k, allora a destinazione si ottiene la sequenza $c_1, c_2, c_3, ..., c_{k'}, ..., c_{n-1'}$ ove $c_{k'} = c_{k+1}, c_{k+1'} = c_{k+2}, ..., c_{k+i'} = c_{k+i+1}$; oppure, con OFB o CBC, se extra caratteri sono aggiunti al flusso cifrato, i.e. se nel flusso

cifrato$c_1, c_2, c_3, ..., c_n, ...$ si aggiunge il carattere $c*$ dopo di c_{k-1}, allora a destinazione si ottiene la sequenza $c_1, c_2, c_3, ..., c_{k'}, ..., c_{n+1'}$ ove $c_{k'} = c*, c_{k+1'} = c_k, ..., c_{k+i'} = c_{k+i-1}$. Quindi l'intera parte restante della trasmissione risulta indecifrabile poiché $m_{i'} = c_{i'} \oplus b_i$ e $b_i = b_i(K, IV)$ cioè bi non dipende dalla sequenza cifrata. Invece, con 8-bit CFB, si ha un effetto risincronizzante: se un byte ci è perso in trasmissione allora corrispondente testo in chiaro mi è perso, e i successivi 8 byte $m_{i+1}, \ldots, m_{i+8}$ risulteranno indecifrabili, ma dal byte m_{i+9} in poi il testo in chiaro sarà corretto, questo perché $b_i = b_i(K, c_{i-1})$, cioè b_i è derivato dalla sequenza di caratteri cifrati. Discorsi analoghi valgono nel caso dell'aggiunta di un byte al flusso cifrato.

- i messaggi cifrati con CFB offrono più protezione di CBC e di OFB rispetto ad eventuali manomissioni; infatti, nel caso di 8-bit CFB un avversario può modificare ogni singolo byte in modo predittivo, ma con l'effetto collaterale di non poter prevedere/controllare i successivi 8 byte discorsi simili valgono per 64-bit CFB.

- a differenza di CBC, non sono possibili attacchi basati sul riarrangiamento di blocchi; tuttavia intere sezioni del messaggio possono essere riarrangiate rendendo indecifrabili le parti corrispondenti ai "punti di giuntura".

CBF ha anche i seguenti svantaggi:

- 8-bit CFB ha lo svantaggio che ogni byte di input richiede un'operazione DES. Inizialmente CFB fu concepito per essere utilizzato con un numero arbitrario k di bit per "pezzo", con k minore della dimensione di un blocco completo (64 bit per DES); nella pratica tuttavia k è pari a 1 byte oppure coincide con la dimensione piena (full-block) dei blocchi del modulo di cifratura. Quando utilizzato in modalità full-block le prestazioni di CFB sono comparabili a quelle di ECB, CBC, e OFB.

- come OFB consente di cifrare ed inviare ciascun byte del messaggio non appena è noto tuttavia, a differenza di OFB non è in grado di anticipare il calcolo del one-time pad in fine, è in grado di rilevare delle alterazioni meglio di OFB, ma non bene quanto CBC.

3.2.6 CounTeR Mode (CTR)

CTR (Figura 3.13) è simile a OFB perché un one-time pad viene generato e sommato ($\oplus$ XOR) con i dati; tuttavia differisce da OFB perché non concatena ciascun blocco di one-time-pad con il precedente, ma incrementa IV e poi cifra quanto ottenuto per ottenere il prossimo blocco di one-time pad.

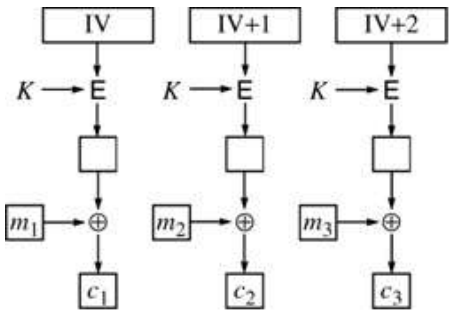


Figura 3.13: CTR

3.2.7 Vantaggi e svantaggi di CTR

Il vantaggio principale di CTR è che, come OFB, il one-time pad può essere pre-calcolato, e la cifratura consiste in un semplice XOR; inoltre, come in CBC, la decifratura di un messaggio può iniziare da un qualunque blocco (non e obbligata ad iniziare dal primo blocco). Per questo CRT

è l'ideale in applicazioni che richiedono la cifratura di file/memorie ad accesso casuale (sottoinsiemi di dati prelevati ed ordine non prevedibili).

Come in OFB (e in tutti i cifrari a flusso), nella modalità CTR si ha una perdita di sicurezza se messaggi diversi sono cifrati con la stessa coppia $\langle K, IV \rangle$, poichè un avversario potrebbe ottenere la somma ($\oplus$ XOR) dei testi in chiaro se somma ($\oplus$ XOR) due testi cifrati ottenuti con la stessa coppia $\langle K, IV \rangle$.

3.3 Generare message authentication code(MAC)

Un sistema di cifratura a chiave segreta può essere usato per generare un MAC cioè un checksum cifrato: **MAC** sta per **M**essage **A**uthentication **C**ode.
Un sinonimo di MAC è **MIC** (**M**essage **I**ntegrity **C**ode) e, anche se il termine MAC è più popolare; nella **PEM** (**P**rivacy **E**nhanced **M**ail) viene usato il termine MIC.

Le modalità operative CBC, CFB, OFB, e CTR offrono una buona protezione della confidenzialità, i.e. un messaggio intercettato è difficilmente decifrabile, ma non offrono una buona protezione dell'integrità/autenticità di un messaggio, i.e. non proteggono da ascoltatori che lo modificano in modo non rilevabile.
Nel seguito useremo i termini integrità e autenticità in modo intercambiabile visto che se il messaggio é integro allora non è stato modificato dal momento in cui è stato generato, i.e. il messaggio è autentico.

3.3.1 Residuo CBC

Un modo standard per assicurare l'autenticità di un messaggio m (cioè per proteggersi da modifiche di m non rilevabili) è, in riferimento alla Figura 3.14 :

- calcolare il CBC di m
- inviare soltanto l'ultimo blocco cifrato (64 bit) e il messaggio m in chiaro; l'ultimo blocco cifrato è detto **residuo CBC**.

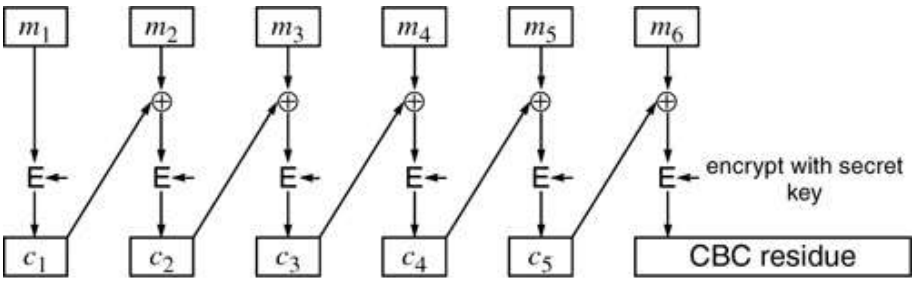


Figura 3.14: Residuo CBC

Il calcolo del residuo CBC richiede la conoscenza della chiave segreta K, quindi se un avversario modifica m in m' allora $res_{CBC}(m')$ sarà diverso da $res_{CBC}(m)$ (c'è solo 1 possibilità su 2^64 che siano uguali) perchè l'avversario non è in grado di calcolare $res_{CBC}(m')$ senza conoscere la chiave segreta K.
Il destinatario del messaggio calcola il residuo del messaggio in chiaro ricevuto, e verifica che sia uguale al residuo ricevuto; se i residui coincidono deduce che (con elevata probabilità) il residuo ricevuto è stato calcolato da qualcuno che conosce la chiave segreta, i.e. il mittente è autentico.

In molte applicazioni non è necessario proteggere la confidenzialità, ma solo l'autenticità; in questi casi si può trasmettere il testo in chiaro più il residuo. Tuttavia, è assai frequente la

necessità di proteggere contemporaneamente confidenzialità e autenticità: se il messaggio m è un singolo blocco, ciò può ottenersi con una semplice cifratura a chiave segreta. Nel caso di un messaggio multi blocco qual è la trasformazione equivalente?

3.3.2 Assicurare confidenzialità e autenticità

Dato un messaggio m:

- per assicurare la **confidenzialità** di m basta cifrarlo in modalità CBC
- per assicurare l'**autenticità** di m basta inviare $res_{CBC}(m)$ più m in chiaro

Allora a prima vista la soluzione riportata in Figura 3.15 sembrerebbe assicurare confidenzialità e autenticità: In realtà tale soluzione è chiaramente errata, infatti consiste nell'inviare il mes-

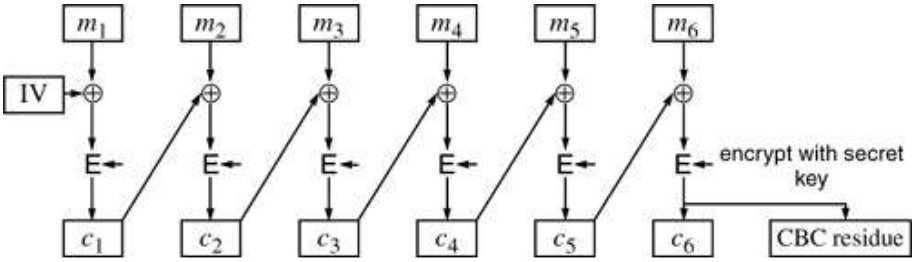


Figura 3.15: Assicurare confidenzialità e autenticità: soluzione 1

saggio cifrato nella modalità CBC ($E_{CBC}(K, IV, m)$) ripetendo soltanto l'ultimo blocco cifrato ($res_{CBC}(m)$); così chiunque voglia alterare il messaggio deve solo modificare uno o più blocchi cifrati con CBC e inviare il nuovo messaggio ripetendo due volte l'ultimo blocco cifrato.
Quindi inviare il residuo CBC in aggiunta al messaggio cifrato con CBC non aumenta la sicurezza.
Si noti infatti che per autenticità (integrità) intendiamo che un calcolatore è in grado di rilevare automaticamente se il messaggio è stato alterato. Usando CBC da solo, allora, non è possibile rilevare in modo automatico eventuali modifiche di un messaggio, poiché ogni stringa di bit, comunque venga generata, viene decifrata in "qualcosa", e gli ultimi 64 bit di quella stringa sono il suo residuo CBC corretto; in questo modo chiunque intercetti il testo cifrato può modificarlo, e un computer a destinazione decifrerà il risultato, che potrebbe essere assolutamente privo di senso, senza essere consapevole che quanto ottenuto è di fatto spazzatura.
Un utente umano si renderebbe conto che il testo cifrato è stato alterato, a meno che la modifica non sia stata eseguita da un avversario in modo "pulito", ma un computer non è in grado di farlo se non si aggiunge un controllo di integrità.

Un'alternativa potrebbe essere calcolare il residuo CBC del messaggio $res_{CBC}(m)$, allegarlo al testo in chiaro m, e cifrare con CBC la concatenazione $m|res_{CBC}(m)$ (Figura 3.16) In realtà

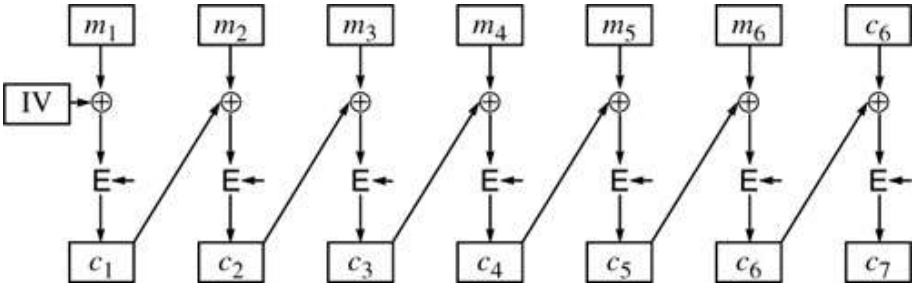


Figura 3.16: Assicurare confidenzialità e autenticità: soluzione 2

neanche questa soluzione funziona, infatti: $c_7 = res_{CBC}(m) = E(K, c_6 \oplus c_6) = E(K, 000\ldots0)$, cioè il residuo CBC è una stringa ottenuta cifrando con la chiave segreta una stringa di 64 bit a 0, quindi il residuo CBC non dipende da m e non può offrire alcune protezione di integrità.

Come altra alternativa, supponiamo di calcolare un checksum non crittografico $CRC(m)$ (ad esempio, un CRC) del messaggio m e di appenderlo alla fine di m, e di cifrare con CBC il tutto: $m|CRC(m)$, secondo lo schema in Figura 3.17 Questa soluzione "quasi" funziona: è vulnerabile

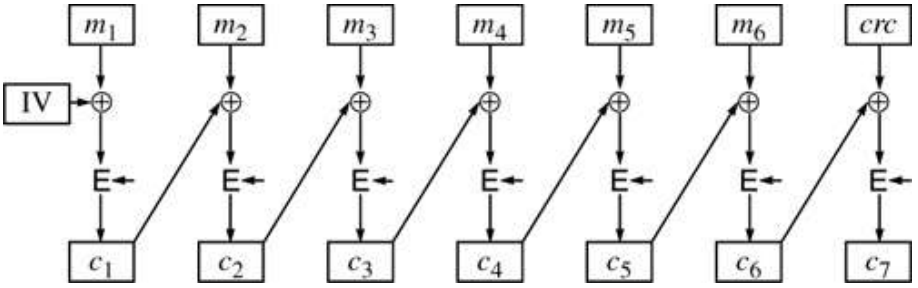


Figura 3.17: Assicurare confidenzialità e autenticità: soluzione 3

ad attacchi molto sottili se il CRC è corto, d'altro canto checksum non crittografici più lunghi sono "sospetti".

Soluzione sicura

Viene considerata una soluzione sicura quella che riesce a proteggere la confidenzialità di un messaggio m cifrandolo con CBC, e l'integrità di m con un residuo CBC, a patto che vengano usate due chiavi distinte, K e K':

- confidenzialità: $E_{CBC}(K, IV, m)$
- integrità: $res_{CBC}(K', IV, m)$

Chiaramente ciò comporta una notevole perdita di efficienza, infatti il costo computazionale è duplicato rispetto al costo della sola cifratura CBC.
Sono state proposte tecniche più rapide, ma generalmente presentano sempre dei "sottili difetti" crittografici (se tali difetti siano seri o meno dipende dal tipo di applicazione e dall'intelligenza dell'avversario); alcune di queste tecniche sono:

- **CBC con un Checksum Crittografico Debole**: Visto che l'uso di checksum non crittografici in CBC risulta poco sicuro, e che checksum crittografici di qualità sono computazionalmente dispendiosi, è stato proposto di usare checksum crittografici "deboli". Complessivamente dovrebbe essere una soluzione sicura, infatti lo sforzo computazionale per violare il checksum debole va moltiplicato per le limitazioni derivanti dal fatto che è usato in una cifratura CBC. Sebbene non ci sono ragioni per sostenere che tale schema sia insicuro, CBC con checksum crittografici deboli non ha riscosso successo (si consideri che Kerberos IV usa un checksum crittografico debole per la protezione d'integrità fuori da uno schema di cifratura e sembra che non sia mai stato violato!)
- **Cifratura CBC e Residuo CBC con Chiavi Correlate**: anziché usare due chiavi completamente indipendenti per la cifratura CBC e per il calcolo del residuo, un trucco usato in Kerberos V è impiegare una versione modificata della chiave in una delle due operazioni. Cambiare un singolo bit dovrebbe essere sufficiente, ma Kerberos invece somma ($\oplus$ XOR) la chiave con la costante a 64 bit $F0F0F0F0F0F0F0F0_{16}$; tale soluzione preserva la parità della chiave e non trasforma mai una chiave non-debole in una chiave debole. Il fatto di avere una chiave matematicamente correlata all'altra (in alternativa alla scelta di due numeri random come chiavi) non introduce particolari debolezze, ma non introduce

neanche particolari vantaggi (in generale, distribuire una coppia di chiavi non è più difficile di distribuirne solo una, e il fatto che due chiavi siano matematicamente correlate non riduce il carico computazionale, l'unico vantaggio nel derivare una chiave dall'altra lo si ha quando si dispone di un sistema/servizio per la distribuzioni di chiavi singole che non è estendibile al caso di coppie di chiavi).

- **CBC con Hash Crittografico**: un altro approccio è concatenare un messaggio m con il suo un hash crittografico $h(m)$, tipicamente 128 bit, e cifrare con CBC il tutto, $m|h(m)$. Tale soluzione è probabilmente sicura, sebbene non sia stata adeguatamente studiata, visto che gli schemi moderni usano hash cifrati con chiavi richiede due fasi crittografiche (come nel caso della cifratura CBC più il residuo CBC con chiavi distinte), ma è più efficiente se la funzione di hash è più veloce dell'algoritmo di cifratura
- **Offset Codebook Mode (OCB)**: OCB è uno dei molti modi che permette di ottenere cifratura e protezione di integrità effettuando soltanto una singola fase di cifratura. OCB e altre tecniche simili sono molto recenti per avere un supporto di studi e test, spesso sono gravate da licenze/patenti, ma sembra molto probabile che una o più di queste tecniche diventi alla fine il modo standard per ottenere protezione di integrità e di confidenzialità.

3.4 Cifratura multipla DES

3.4.1 Cifratura multipla EDE o 3DES

In generale, ogni schema di cifratura può essere reso più sicuro ricorrendo alla cifratura multipla. Nel caso di DES, è universalmente ritenuta sicura la procedura nota come EDE. **E**ncrypt-**D**ecrypt-**E**ncrypt (o 3DES, **triplo DES**):

- $c = E(K_1, D(K_2, E(K_1, m))) = (E_1 o D_2 o E_1)(m)$
- $m = D(K_1, E(K_2, D(K_1, c))) = (D_1 o E_2 o D_1)(c)$

La cifratura multipla EDE è di fatto un meccanismo per incrementare la lunghezza della chiave DES. Sebbene sia stata introdotta per arrivare ad uno standard sicuro basato su DES, in linea di principio è applicabile anche ad altri schemi di cifratura, ed esempio ad IDEA, ma è più importante nel caso di DES poiché la chiave DES è notoriamente considerata troppo corta.

Si è visto che uno schema di crittografia presenta sempre due funzioni, note come cifratura e decifratura; tali funzioni sono l'una l'inversa dell'altra, i.e.: ciascuna prende in input un blocco di dati m, e restituisce il corrispondente blocco offuscato c tale che, applicando a c l'altra funzione si ottiene m. Ha quindi senso applicare la "funzione di decifratura" $D(K, m)$ al testo in chiaro m per cifrarlo e poi applicare la "funzione di cifratura" $E(K, D(K, m))$ per decifrare quanto ottenuto riottenendo il testo in chiaro m; ma in definitiva ha poco senso chiamare $E(K, m)$ "funzione di cifratura", e $D(K, m)$ "funzione di decifratura", poiché i ruoli di tali funzioni sono interscambiabili. Conviene chiamarle semplicemente **funzione** $E()$ e **funzione** $D()$ (tuttavia, spesso si continuerà a chiamarle funzione di cifratura e funzione di decifratura).

Realizzare una cifratura multipla non è banale, specie se si desidera anche ottenere un cifrario a flusso a partire da un cifrario a blocchi.
Il metodo standard per usare EDE è il seguente: vengono usate due chiavi (e non tre), K_1 e K_2 e ogni blocco di testo in chiaro m_i è sottoposto prima ad $E_1()$ (cioè viene eseguito $E(K_1, m)$), poi a $D_2()$ (cioè viene eseguito $D(K_2, E(K_1, m))$) e in fine ancora ad $E_1()$ (cioè viene eseguito $E(K_1, D(K_2, E(K_1, m)))$).
Quanto ottenuto è di fatto un nuovo schema di cifratura a chiave segreta (Figura 3.18): un blocco di 64 bit in input è mappato in un altro blocco di 64 bit in output; il processo è invertibile se si conoscono le chiavi, altrimenti è di fatto impraticabile risalire dall'output all'input. La decifratura EDE è semplicemente il processo inverso (Figura 3.19):

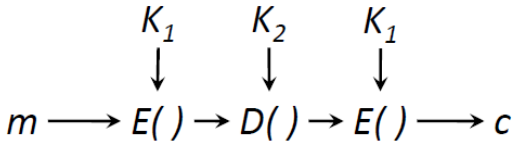


Figura 3.18: Cifratura EDE o 3DES

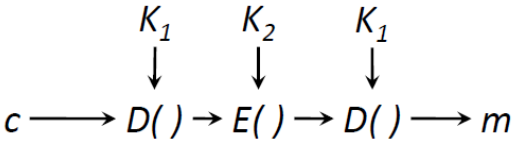


Figura 3.19: Decifratura EDE

3.4.2 EDE con CBC Outside/Inside

Per ottenere un cifrario a flusso (i.e. la corrispondente modalità operativa di flusso) a partire dal cifrario a blocchi EDE, viene usata la modalità operativa **CBC outside**, i.e. le tre funzioni $E_1() - D_2() - E_1()$ vengono applicate a ciascun blocco, ma il concatenamento CBC viene eseguito soltanto una volta (Figura 3.20).

Il cifrario a flusso è ottenibile anche con un concatenamento **CBC inside**, i.e. si considerano una

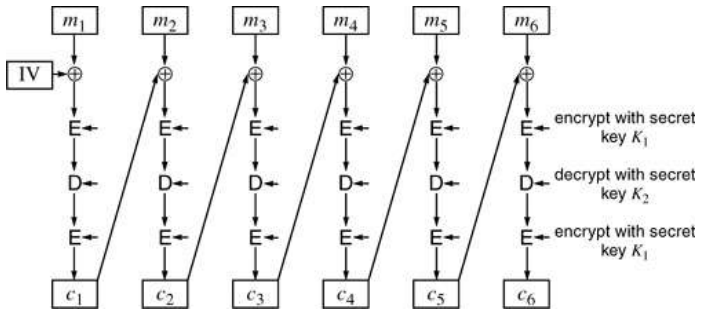


Figura 3.20: EDE con CBC Outside

cascata di tre concatenamenti CBC (semplificando alcuni passaggi): prima utilizzando $CBC - E_1()$, poi utilizzando $CBC - D_2()$ e infine utilizzando ancora $CBC - E_1()$ (Figura 3.21).

CBC Outside vs Inside

Il 3DES comunemente usato nelle applicazioni esegue un concatenamento CBC esterno: ad ogni blocco viene applicata la cifratura tripla e il concatenamento CBC viene fatto una sola volta sui blocchi cifrati. L'alternativa sarebbe cifrare completamente il messaggio con K_1 e CBC, poi decifrare il risultato con K_2 e CBC, e in fine cifrare di nuovo quanto ottenuto con K_1 (CBC interno).
Quali sono le implicazioni di queste scelte?

Si è visto che con CBC è possibile fare una modifica predittiva sul testo in chiaro m_n (ad esempio invertire il bit x) invertendo il bit x nel blocco cifrato c_{n-1}; ciò comporta però l'effetto collaterale di modificare in modo imprevedibile il blocco m_{n-1}, la possibilità di sfruttare questa debolezza dipende dal tipo di applicazione.

Con CBC esterno, un avversario può ancora sferrare questo tipo di attacco (il fatto che la cifratura è fatta con un triplo DES è del tutto ininfluente): un avversario che inverte il bit x di un blocco cifrato c_{n-1} modificherà completamente e in modo imprevedibile il blocco di testo in

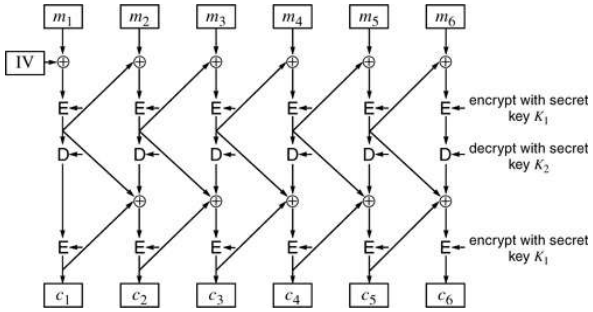


Figura 3.21: EDE con CBC Inside

chiaro m_{n-1} (il blocco di testo in chiaro m_n avrà il bit x invertito e tutti i blocchi di testo in chiaro diversi da m_{n-1} e m_n saranno invariati.

Con CBC interno, una modifica ad un blocco cifrato c_n altera in modo imprevedibile tutti i blocchi di testo in chiaro dal blocco m_n fino alla fine del messaggio. Per questo CBC interno è più sicuro di CBC esterno, e forse dovrebbe essere la scelta migliore; tuttavia, in alcuni casi è preferibile che la modifica di un blocco cifrato non si propaghi completamente nel resto del messaggio, i.e. sarebbe preferibile che lo schema di cifratura sia autosincronizzante (dopo un piccolo numero di blocchi corrotti, il testo in chiaro inizierà ad essere nuovamente decifrato correttamente).
Un altro vantaggio di CBC interno riguarda l'efficienza: triplicando l'uso di hardware e pipeline per le cifrature si può ottenere complessivamente una velocità pari a quella di una singola cifratura (con CBC esterno ciò non è possibile).
CBC interno presenta comunque delle sottili vulnerabilità se un avversario può esaminare l'output e fornire del testo in chiaro scelto e IV.

Una ragione per la quale CBC esterno è più usato nonostante i suoi svantaggi è che la cifratura EDE può considerarsi a tutti gli effetti un nuovo schema di cifratura (a chiave segreta) a blocchi che usa una chiave di 112 bit, perciò può essere utilizzata con ciascun metodo di concatenamento (OFB, ECB, CFB, CTR e CBC).

Capitolo 4

Funzioni Crittografiche di Hash

4.1 Introduzione

Una funzione di hash (o semplicemente hash o message digest) è una funzione unidirezionale (o one-way).

- è una funzione perché prende in input un messaggio e produce un output che dipende dal messaggio :
  - h: $\{0,1\}^* \rightarrow \{0,1\}^b$
  - $\{0,1\}^*$: spazio delle stringhe binarie di lunghezza qualsiasi
  - $\{0,1\}^b$: spazio delle stringhe binarie di lunghezza b bit
  - è considerata unidirezionale (one-way) perché è impraticabile capire quale input corrisponda ad un dato output

  Sia h() una funzione di hash, allora h() è una funzione di hash sicura se :
  - resistenza alla preimmagine: fissato un hash $\hbar$ è computazionalmente impraticabile trovare un messaggio m tale che $h(m) = \hbar$
  - resistenza alle collisioni: è computazionalmente impraticabile trovare due messaggi m1 e m2 aventi lo stesso digest h(m1) = h(m2), le proprietà precedenti implicano la seguente
  - resistenza alla seconda preimmagine: dato un messaggio m, è computazionalmente impraticabile trovare un messaggio m' avente lo stesso digest $h(m) = h(m')$

Si useranno i termini hash e message digest in modo intercambiabile; la funzione di hash del NIST è chiamata SHA-1: Secure Hash Algorithm mentre l'acronimo MD degli algoritmi MD2, MD4 e MD5 sta per Message Digest. Tutti gli algoritmi di digest/hash basicalmente fanno la stessa cosa: prendono in input un messaggio di lunghezza variabile, e restituiscono in output una quantità avente lunghezza prefissata. Dato un messaggio m, il digest h(m) viene calcolato in modo deterministico. Tuttavia, l'output della funzione di hash dovrebbe apparire il più possibile casuale. Dovrebbe essere impossibile, senza applicare la funzione di hash, predire ogni porzione dell'output e per ogni sottoinsieme (di posizioni) di bit nel digest h(m) soltanto procedendo in modo esaustivo dovrebbe essere possibile ottenere due messaggi m1 e m2 tali che h(m1) e h(m2) presentino gli stessi bit in quelle posizioni. Perciò una funzione di hash sicura con n bit dovrebbe essere derivabile da una funzione di hash con più di n bit prendendo un arbitrario sottoinsieme di n bit dal digest più grande. Chiaramente, ci sono molti messaggi distinti che sono mappati in uno stesso digest h(m); m ha lunghezza arbitraria, mentre il digest h(m) ha una lunghezza prefissata, ad esempio 128 bit. Se m ha una lunghezza di 1000 bit e h(m) di 128 bit ci sono in media 2872 messaggi che sono mappati in uno stesso digest dopo molti tentativi, due messaggi aventi lo stesso digest si trovano sicuramente. Tuttavia, per "molti tentativi" si intende un numero talmente grande che è di fatto impossibile considerando una buona funzione di digest a 128 bit, è

necessario provare approssimativamente 2^{128} possibili messaggi prima di ottenere un messaggio avente un particolare digest, o 2^{64} messaggi prima di trovarne due aventi lo stesso digest ; trovare cioè due messaggi che collidono.

4.2 Esempio di Applicazioni

Un'applicazione delle funzioni di hash crittografiche è il calcolo dell'impronta digitale di un programma o di un documento di cui si desiderano monitorare eventuali modifiche : se il message digest h(p) del programma p è noto e se h(p) è memorizzato in modo sicuro (cioè non può essere modificato da utenti non autorizzati) allora nessun utente non autorizzato può modificare p senza essere scoperto perché non sarà in grado di trovare un diverso programma p' tale che h(p') = h(p). Quindi sia h:$\{0,1\}^* \rightarrow \{0,1\}^b$ una funzione di hash siano $m_1, m_2, ..., m_N$ N messaggi arbitrariamente scelti in $0, 1^*$.
DOMANDA: quanto deve valere N per avere una probabilità di 0.5 che due messaggi m_i, m_j abbiano lo stesso hash?

Una prima stima di N (affinché ci sia 0.5 di possibilità di collisione), per difetto, è la seguente :

- $k = 2^b$: numero totale di possibili hash
- $\frac{1}{k}$: probabilità che una coppia di messaggi collida
- **Ipotesi**: gli eventi considerati sono mutuamente esclusivi
  - $\Pr\{h(mi) = h(mj) OR h(mp) = h(mq)\} = \Pr\{h(mi) = h(mj)\} + \Pr\{h(mp) = h(mq)\}$

A rigore tale ipotesi non è soddisfatta, molti eventi hanno intersezione non nulla, perciò la stima ottenuta della probabilità è per eccesso e ciò si traduce in una stima per difetto di N ($\Pr\{E1 ORE2\} = \Pr\{E1\} + \Pr\{E2\} \breve{} \Pr\{E1 \bigcap E2\}$). Si ha una probabilità di 0.5 se si considerano $k/2$ coppie, perciò $N(N \breve{} 1)/2 = k/2$ e ipotizzando $N >> 1$ si ha $N = k^{1/2} = 2^{b/2}$.

Una seconda stima, più accurata di N è la seguente:

- P: probabilità che almeno una coppia di messaggi collida
- P^*: probabilità che tutte le coppie di messaggi abbiano digest diversi, quindi $P = 1 - P^*$
- **Ipotesi**: gli eventi considerati sono indipendenti
  - $\Pr\{h(mi) \neq h(mj) AND h(mp) \neq h(mq)\} = \Pr\{h(mi) \neq h(mj)\} * \Pr\{h(mp) \neq h(mq)\}$

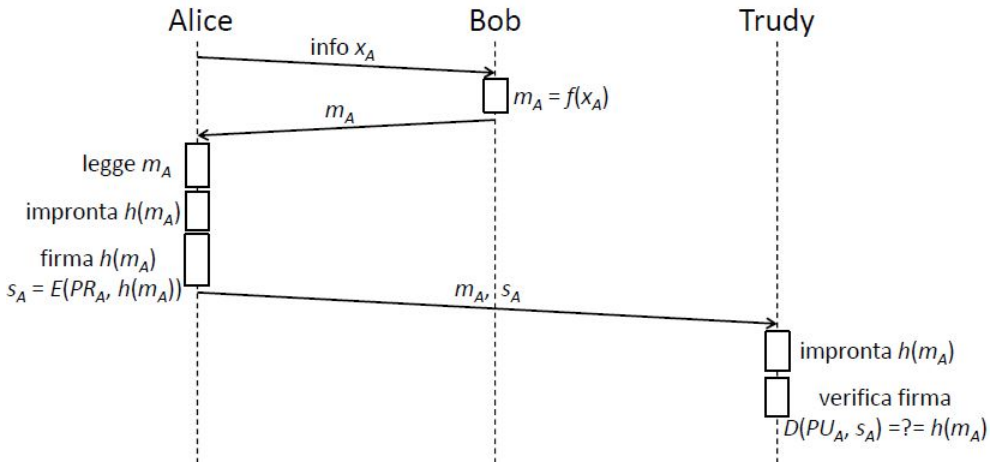
A rigore tale ipotesi non è soddisfatta, gli eventi non sono del tutto indipendenti quindi la stima ottenuta della probabilità P è per difetto e ciò si traduce in una stima per eccesso di N ($\Pr\{E1 AND E2\} = \Pr\{E1\} * \Pr\{E1|E2\}$). Nella ipotesi di eventi indipendenti si ha $P = 1 \breve{} P^* = 1 \breve{} (1 \breve{} 1/k)^{\frac{N(N \breve{} 1)}{2}}$, e ipotizzando che k » 1 si ottiene che $P \approx 1 \breve{} e^{\breve{} N(N \breve{} 1)/2k}$, ponendo pertanto $P \geq 1/2 siottienechee^{\breve{} N(N \breve{} 1)/2k} \leqslant 1/2 = e^{\breve{} ln2}$; da cui si ottiene che $N(N \breve{} 1)/2 \geq (ln2)k$ e, ipotizzando che N » 1, $N \geqslant (2ln2)^{1/2} \cdot k^{1/2} = (2ln2)^{1/2} \cdot 2^{b/2}$.

Una terza stima ancor più corretta si può ottenere utilizzando la modalità di calcolo utilizzata anche nel paradosso del compleanno :

- P: probabilità che almeno una coppia di messaggi collida
- P^*: probabilità che tutte le coppie di messaggi abbiano digest diversi, quindi $P = 1 - P^*$
- $k = 2^b$: numero di possibili hash

Si consideri inoltre la seguente notazione:

- P_{2^*}: la probabilità che h(m2) sia diverso da h(m1)
- P_{3^*}: la probabilità che h(m3) sia diverso da h(m2) e h(m1)



- ...
- P_{i^*}: la probabilità che h(mi) sia diverso da h(mj), $\forall 1 \leq j \leq i-1$
- ...
- P_{N^*}: la probabilità che h(mN) sia diverso da h(mj), $\forall 1 \leq j \leq N-1$

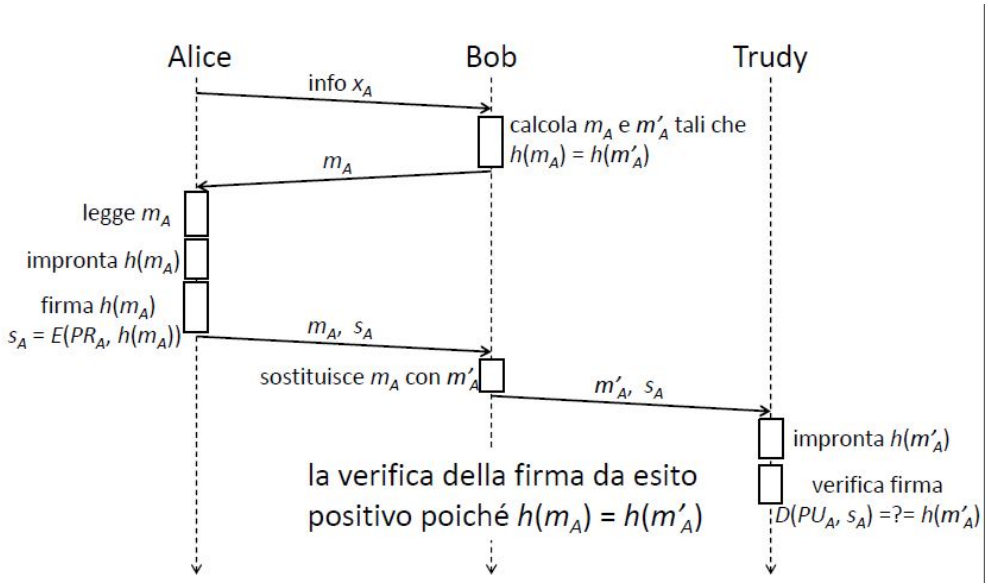
Da tutto ciò segue che $P^* = P_{2^*} * P_{3^*} * ... * P_{i^*} * ... * P_{N^*} = (k \breve{} 1)/k * (k \breve{} 2)/k * ... * (k \breve{} N+1)/k = \frac{k!}{(k^N * (k \breve{} N)!)} \Rightarrow P^* = \frac{k!}{(k^N * (k \breve{} N)!)}$. Questa è la stima esatta di P^*, si nota che il precedente valore di P^* si può approssimare con $1 \breve{} e \breve{}^{N(N \breve{} 1)/2k}$.

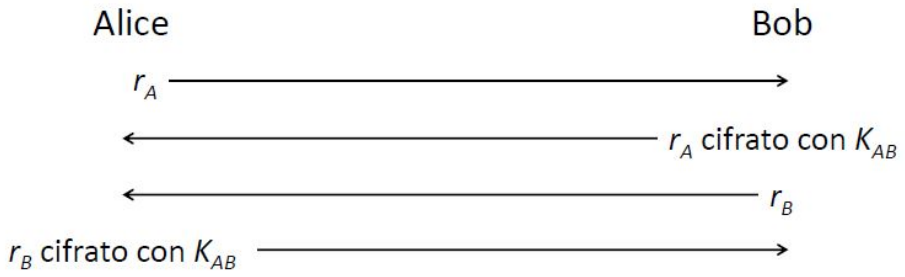
4.3 Lunghezza di un messaggio di Digest

Quanti bit deve avere l'output di una funzione di hash in modo tale che nessuno sia in grado di trovare due messaggi aventi lo stesso digest? Se il digest ha b bit, per trovare due messaggi aventi lo stesso digest, è necessario considerare circa $2^{b/2}$ messaggi, se il digest è lungo 64 bit, la ricerca esaustiva in uno spazio di circa 2^{32} elementi può essere fattibile mentre se il digest è lungo 128 bit si ritiene che una ricerca in uno spazio di 2^{64} elementi sia impraticabile dato l'attuale stato dell'arte. Per quale ragione è importante che una funzione crittografica di hash sia resistente alle collisioni? Il fatto che debba essere resistente alla preeimmagine è scontato, ma la resistenza alle collisioni è veramente necessaria?? Si, dato che in alcune circostanze riuscire a trovare due messaggi con lo stesso digest può comportare dei seri problemi di sicurezza!! Ad esempio, Alice genera una informazione x_A e incarica Bob di calcolare un messaggio m_A il cui contenuto deve dipendere da x_A secondo criteri prestabiliti. Una volta che Bob ha calcolato m_A lo sottopone ad Alice che ne verifica l'integrità e calcola l'impronta $h(m_A)$, la firma con la sua chiave privata e invia a Trudy il messaggio in chiaro m_A e la sua firma. Trudy legge il messaggio m_A e la firma, controlla che la firma sia quella di Alice e verifica che tutto m_A coincida con $h(m_A)$. Se la funzione non è resistente alla collisioni, Bob potrebbe trovare m_{A1} tale che risulti $h(m_A) = h(m_{A1})$. Una volta che Alice ha firmato il messaggio, sostituisce m_A con m_{A1} e Trudy non si potrà mai accorgere di nulla. Ma, per calcolare due messaggi con lo stesso hash, Bob può seguire il seguente approccio a forza bruta:

- dato xA, calcola prima il messaggio mA come desiderato da Alice;
- genera un messaggio falsato $mA^{'}$;
- esegue il test $h(mA) == h(mA^{'})$;
- in caso negativo ritorna al punto 2;

con questo approccio a forza bruta, però, sono necessari circa 2^b tentativi, e non $2^{b/2}$!



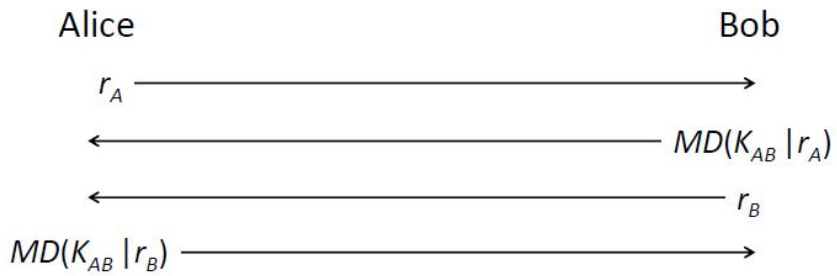


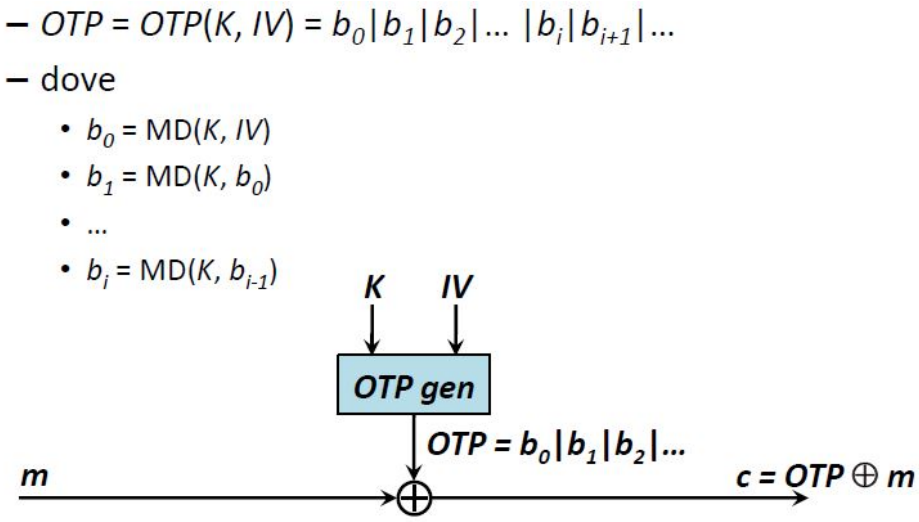
4.4 Impieghi degli Algoritmi di Hash

Disponendo di un segreto condiviso, l'algoritmo di hash può "sostituire" un algoritmo di crittografia a chiave segreta in ogni suo impiego. (Autenticazione a chiave segreta, Calcolo di MAC, Cifratura e Decifratura).

4.4.1 Message Digest per MAC

Un possibile schema di autenticazione a chiave segreta (composta da "sfide") è il seguente Questo schema, purtroppo, presenta delle vulnerabilità...così come il seguente: Le vulnerabilità degli algoritmi di digest MD, sono note e facilmente attuabili in un possibile attacco. Infatti, MD(m) è calcolabile da tutti coloro che conoscono m e l'algoritmo di digest MD senza conoscere la chiave segreta!! Poiché, dall'input m si ottiene un messaggio mp avente lunghezza pari ad un multiplo intero di 512 bit ()mediante un opportuno padding che include, tra l'altro, la lunghezza originaria di m), mp viene decomposto in chunk (pezzi) da 512 bit il digest viene ottenuto mediante una procedura iterativa, quindi il digest all'n-esima iterazione dipende esclusivamente dall'n-esimo chunk e dal digest ottenuto all'(n – 1)-esima iterazione...il digest risultante di m è il digest ottenuto all'ultima iterazione. Se un attaccante intercetta $\langle m, MD(K_{AB}|m)\rangle$ tra Alice e Bob, l'attaccante senza conoscere la chiave segreta K_{AB} può calcolare il MAC di $\langle m^+, MD(K_{AB}|m^*)\rangle$ (dato che l'algoritmo di hash è noto) tramite l'ultimo messaggio intercettato.
Una possibile soluzione è che il segreto K_{AB} sia messo in coda, a patto che l'algoritmo di hash sia molto resistente alle collisioni!! Un altra possibilità è quella di utilizzare come MAC un sot-





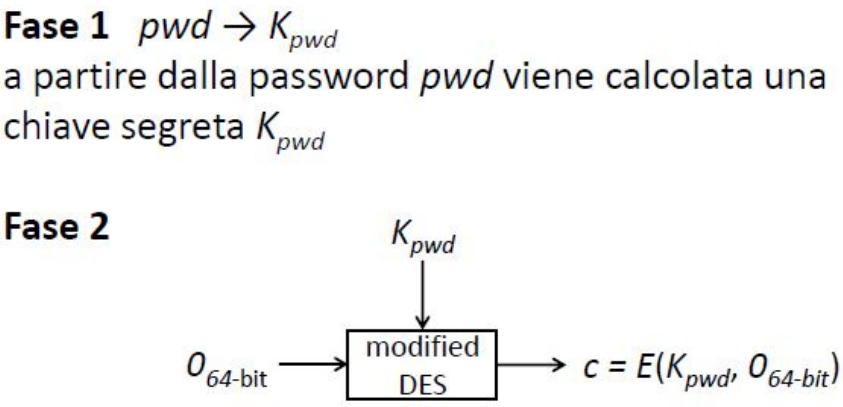
toinsieme arbitrario del digest MD, in questo caso, se il mio MAC è di soli 64 bit (invece di 128), l'attaccante potrebbe sempre ricostruire un messaggio da accodare...ma ha una possibilità su 2^{64} che il MAC ottenuto sia quello corretto.
Una terza soluzione è quella di inserire il segreto K_{AB} sia all'inizio che in coda del messaggio da mandare al digest MD, così che K_{AB} in testa fornisca resistenza alle collisioni e K_{AB} rende innocua la vulnerabilità insita negli algoritmi di hash. Dato che ognuna delle soluzioni è equivalentemente corretta per ovviare al problema della vulnerabilità della funzione MD, per dare una piattaforma condivisa...una soluzione condivisibile da tutti, è stato introdotto il framework **HMAC** (keyed-Hash Message Authentication Code); l'algoritmo HMAC calcola due volte il digest del messaggio, e ogni volta inserisce in testa il segreto K_{AB}.

4.4.2 Cifratura/Decifratura

Come faccio a cifrare con un algoritmo di hash? Devo utilizzare la funzione di hash potendo rendere tutto il processo invertibile, quindi introducendo lo XOR tra un chunk del messaggio e l' hash della chiave (genero un keypad) e ottengo un cifrario a flusso!!

4.4.3 Algoritmi di cifratura come algoritmi di hash

Se un algoritmo di hash/digest, può essere utilizzato come un algoritmo di cifratura...può accadere il contrario? In genere, questo principio è utilizzato (ad esempio) per la memorizzazione degli hash delle password in UNIX. Ma come si modifica un algoritmo di cifratura che sia resistente alle collisioni?
Lo schema di funzionamento (minimale) è questa Ma questo schema funziona per soli messaggi



corti e non è troppo resistente alle collisioni, sebbene K_{PWD} sia ottenuta dalla pwd considerandone i primi 8 caratteri ed espansa con con i bit di parità per ogni gruppo di 8 bit. Dato s il *salt*, invero il numero di 12 bit ottenuto in modo pseudo randomico dalla pwd di ogni utente, la funzione h(pwd) si ottiene dal DES modificato che dipende dal valore di s (che determina quali bit devono essere replicati nella fase di espansione di R da 32 a 48 bit). Infine il DES modificato è utilizzato con K_{PWD} pr cifrare una costante 0 a 64 bit. Il risultato della cifratura e il salt, sono memorizzati per poter recuperare pwd.

4.4.4 Hash di grandi messaggi con algoritmi di cifratura come hash

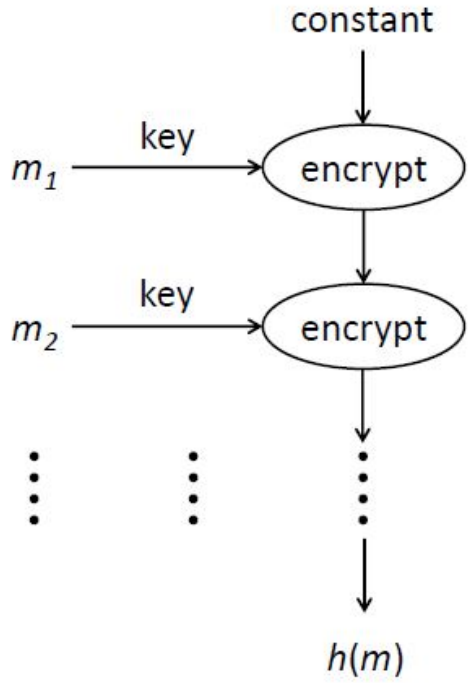
Per l'algoritmo precedente, era stato sottolineato che era valido esclusivamente per messaggi corti. Come si deve operare per estendere questi algoritmi a messaggi di lunghezza arbitraria? Si può pensare di suddividere il messaggio $m = m_1, m_2, m_3, ..., m_n$, ogni m_i viene utilizzato come input a un blocco in cascata di cifratura, alla fine della cascata ho realizzato un algoritmo di hash tramite cifratura...per un messaggio di lunghezza qualsiasi.

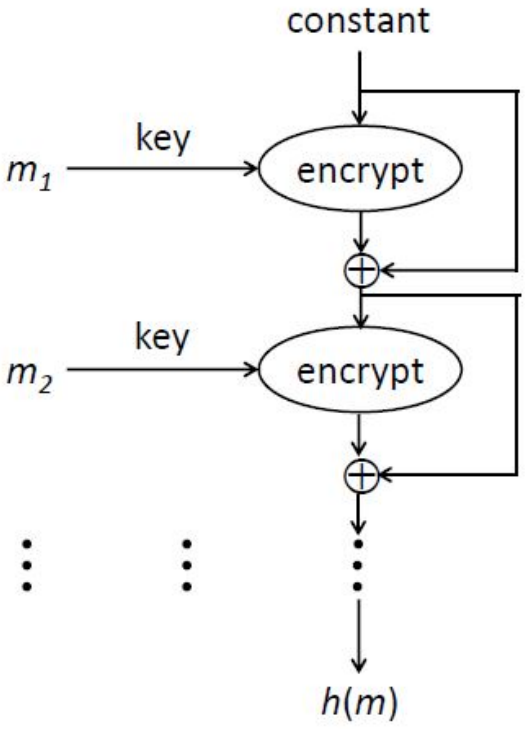
Purtroppo questo sistema soffre degli stessi problemi di cui soffriva il DES doppio, per ovviare ciò, basta rendere l'output di ogni blocco differente diverso dall'input del successivo, una possibile implementazione è la seguente

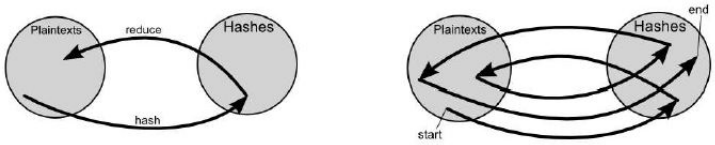
4.4.5 Rainbow Tables (Opzionale)

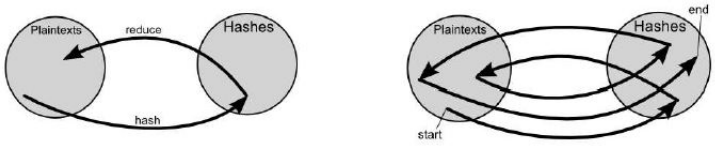
Le Rainbow tables sono un meccanismo per l'attacco di reverse hash. Le tables bilanciano memoria e sforzo computazionale, se fosse utilizzata solo memoria si avrebbero dei database troppo grandi per poter mappare tutte le immagini degli hash e se fosse utilizzata solo potenza computazionale servirebbe troppo tempo per poter trovare l'inverso di un hash.

Le Rainbow tables utilizzano delle funzioni di riduzione r(), la funzione di riduzione è una applicazione dallo spazio degli Hash allo spazio dei Plaintext, r() non è l'inversa di un algoritmo di hash; per questo, prima di avvicinarmi a una buona inversione dovrò concatenare fino a centinaia di iterazioni di $h(m) = k \longrightarrow r(k) = m^{'}$. Per popolare la table, si memorizzano solo il messaggio iniziale e il messaggio finale della catena. In questo modo, con una catena di 100 iterazioni (dato che le funzioni sono deterministiche) si risparmia uno spazio in memoria di 100 volte la grandezza della table, in confronto a memorizzare solo coppie $\langle plaintext, hash \rangle$. Per recuperare un plaintext di un hash, si controlla se questo sia un endpoint di una qualsiasi delle catene (fase di lookup), se non lo fosse si controllano gli endpoint intermedi di ogni catena. Appena recuperato l'hash giusto, si utilizza la catena determinata dall'inizio finchè non si trova l'hash di inizio; il messaggio che ha generato quell'hash sarà ciò che si cercava.


constant
key
m_1
encrypt
key
m_2
encrypt
$h(m)$


constant
key
m_1
encrypt
key
m_2
encrypt
$h(m)$


Plaintexts
reduce
Hashes
hash
Plaintexts
Hashes
end
start


Plaintexts
reduce
Hashes
hash
Plaintexts
Hashes
end
start

Capitolo 5

Crittografia a chiave pubblica

La crittografia a chiave pubblica si basa su alcuni risultati nell'ambito della teoria dei numeri. Si esamineranno i seguenti schemi di cifratura a chiave pubblica:

- RSA usata per cifrare e per calcolare la firma digitale
- ElGamal e DSS, usati per la firma digitale
- Diffie-Hellman, permette di stabilire un segreto condiviso, ma non fornisce alcun algoritmo che usa effettivamente tale segreto

L'unico aspetto comune a tutti gli algoritmi di crittografia a chiave pubblica è la presenza di due quantità correlate: una chiave segreta e una chiave pubblica.

5.1 Aritmetica modulare

La maggior parte degli algoritmi a chiave pubblica si basano sull'aritmetica modulare: fissato un intero $n > 1$, l'aritmetica modulare considera l'insieme degli interi non negativi minori di n :$\{0, 1, 2, ..., n-1\}$, effettua operazioni ordinarie come l'addizione e la moltiplicazione, e sostituisce il risultato x con il resto r della divisione intera di x per n. Il risultato finale viene detto modulo n o $mod\, n$.
Definiamo l'inverso moltiplicativo di k, indicato con k^{-1}, come quel numero che moltiplicato per k dà 1, cioè $kk^{-1}\, mod\, n = 1\, mod\, n$. Fissato n, non tutti i numeri hanno un inverso moltiplicativo $mod\, n$. Si osservi inoltre che, se k ammette un inverso moltiplicativo $mod\, n$, esiste un unico inverso moltiplicativo $k^{-1} < n$. La moltiplicazione $mod\, n$ di per sé non costituisce un cifrario sicuro,ma funziona, nel senso che la moltiplicazione per k produce un mescolamento dell'input; la decifratura può ottenersi moltiplicando per k^{-1}.
Trovare un inverso moltiplicativo k^{-1} nella aritmetica $mod\, n$, non è affatto banale se n è molto grande. Esiste un modo efficiente per risolvere tale problema, noto come algoritmo di Euclide:

- dati x ed n, con $x < n$, l'algoritmo di Euclide trova il numero $y < n$ tale che $xy\, mod\, n = 1$, ammesso che un siffatto y esista.

Quali sono dunque gli inversi moltiplicativi $mod\, n$? E' sufficiente trovare i numeri *relativamente primi* con n, cioè tali che se x è uno di questi numeri, si ha che $MCD(x, n) = 1$. Se n è un numero primo, tutti gli interi positivi $x < n$ ammettono un inverso moltiplicativo $mod\, n$, che indichiamo con x^{-1}.

Funzione di Eulero o totiente: $\{i \in Z, 0 < i < n : MCD(i, n) = 1\}$

Se n è primo, tutti gli interi da 1 a $n-1$ sono relativamente primi con n, per cui $\phi(x) = n-1$. Se $n = pq$, dove p e q sono numeri primi maggiori di 1, allora $\phi(n) = (p-1)(q-1)$.
Sia adesso $n > 1$ un intero privo di quadrati(cioè dove non compaiono fattori al quadrato nella sua scomposizione in fattori primi), detto anche square free, allora per ogni $y > 0$, si ha:

$$x^y \bmod n = x^{(y+\phi(n))} \bmod n$$

Ne segue che, se $y = 1 \bmod \phi(n)$, ovvero se $y = 1 + k\phi(n)$, con $k \in Z$, si ha $x^y \bmod n = x \bmod n$. Tale risultato è sfruttato dall'algoritmo RSA.

5.2 RSA

RSA è un algoritmo di cifratura(a blocchi) a chiave pubblica, la cui lunghezza è variabile(solitamente si considerano chiavi di lunghezza pari ad almeno 512 bit). Anche la lunghezza dei blocchi è variabile: un blocco di testo in chiaro deve avere lunghezza minore di quella della chiave, mentre un blocco di testo cifrato è lungo come la chiave.
RSA è computazionalmente molto più lento degli algoritmi a chiave segreta più popolari come DES, IDEA e AES, per cui difficilmente viene usato per cifrare messaggi lunghi. Generalmente viene usato per cifrare una chiave segreta K, utilizzata per cifrare un messaggio usando un algoritmo a chiave segreta. RSA può essere usato dunque sia per cifrare/decifrare messaggi sia per la firma digitale di messaggi. In entrambi i casi bisogna disporre della coppia $< chiave\,pubblica, chiave\,privata >$ ($< PU, PR >$).
I passi da seguire per generare la chiave pubblica e la chiave privata sono:

1. Scegliere due numeri primi p e q molto grandi(circa 256 bit ciascuno) tali che $n = pq$. E' fondamentale che p e q rimangano segreti, cosicchè fattorizzare n sia computazionalmente impraticabile.
2. Scegliere un numero e che sia relativamente primo rispetto a $\phi(n) = (p-1)(q-1)$.
3. Calcolare l'inverso moltiplicativo d di $e \bmod \phi(n)$, cioè tale che sia $(d \cdot e) \bmod \phi(n) = 1$.
4. La chiave pubblica è $PU =< e, n >$, mentre la chiave privata è $PR =< d, n >$.

Per quanto riguarda la **cifratura/decifratura**, siano PU e PR la chiave pubblica e la chiave privata del destinatario, m il messaggio da cifrare. La procedura da seguire è la seguente:

- il mittente, utilizzando la chiave pubblica PU del destinatario, cifra il messaggio ottenendo $c = m^e \bmod n$
- il destinatario, usando la propria chiave privata PR, decifra c calcolando $m = c^d \bmod n$.

Dimostrazione. Poste le seguenti proprietà:

- se $m < n,\ m \bmod n = n$
- $(x^a \bmod n)^b \bmod n = x^{ab} \bmod n$
- $(e \cdot d) \bmod \phi(n) = 1$

Se $c = E(PU, m) = m^e \bmod n \Rightarrow m = D(PR, c) = c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{ed} \bmod n = m^{1 \bmod \phi(n)} \bmod n = m \bmod n = m$.

Per la **firma digitale** invece sia PR la chiave privata del firmatario del messaggio e sia PU la sua chiave pubblica:

- il firmatario, usando la propria chiave privata PR, calcola la firma digitale $s = m^d \bmod n$;
- chiunque desideri verificare l'autenticità della firma, può farlo usando la chiave pubblica PU del firmatario e calcolando $m = s^e \bmod n$.

La dimostrazione è per la firma è analoga alla cifratura.

La sicurezza di RSA deriva dal fatto che fattorizzare interi molto grandi è impraticabile. Infatti, identificando i numeri primi p e q tali che $n = pq$, si ottiene $\phi(n) = (p-1)(q-1)$ e quindi si può calcolare d come l'inverso moltiplicativo di $e \bmod \phi(n)$, ottenendo la chiave privata

$PR =< d, n >$ dalla chiave pubblica $PU =< e, n >$.
Tuttavia è possibile violare RSA senza ricorrere alla fattorizzazione, se lo si usa in modo improprio.
In base al tipo di impiego, RSA svolge le seguenti operazioni molto frequentemente (ad ogni sessione di lavoro):

- cifratura/decifratura
- generazione/verifica di una firma digitale

E' necessario pertanto che tali operazioni siano svolte nel modo più efficiente possibile. Invece, l'operazione di generazione delle chiavi viene eseguita meno frequentemente e quindi si può tollerare una minore efficienza.
Le operazioni di cifratura, decifratura, firma e verifica della firma richiedono tutte di dover considerare un intero molto grande, elevarlo ad un esponente(intero) molto grande e trovare il resto della divisione intera per un numero molto grande. Considerando la dimensione dei numeri interi per i quali RSA è ritenuto sicuro, tali operazioni risulterebbero proibitive se eseguite nel modo più ovvio.

5.2.1 Generazione delle chiavi RSA

La generazione delle chiavi RSA è un'operazione poco frequente: in gran parte delle applicazioni della tecnologia a chiave pubblica deve essere eseguita soltanto una volta e non è richiesta la stessa efficienza delle altre operazioni RSA; deve comunque essere garantita un'efficienza ragionevole.
Per generare una coppia di chiavi $< PU, PR >$ è necessario trovare due numeri primi p e q molto grandi e trovare due interi d ed e con le proprietà precedentemente descritte.

Trovare due numeri primi grandi p e q. Esistono infiniti numeri primi, che diminuiscono all'aumentare di n: estraendo un numero a caso, si ha che $Pr\{n\,primo\} \approx 1/ln\,n \approx 1/N_b$, dove N_b è il numero di bit utilizzato per rappresentare n. La densità dei numeri primi è inversamente proporzionale alla loro lunghezza in bit(o in cifre decimali). Ad esempio, per un numero n a cento cifre decimali (dimensione usata in RSA), c'è una possibilità su 230 che esso sia primo.
Pertanto, i passi da seguire per generare p e q sono i seguenti:

1. estrai un numero dispari molto grande
2. verifica se tale numero è primo, in caso negativo ritenta(in media, sono necessari 230 tentativi per ottenere un numero primo)

Tale strategia va bene se si dispone di un test di primalità efficiente: come è possibile testare se un intero n è primo? Un metodo banale consiste nel dividere n per tutti gli interi $\leq n^{1/2}$ e verificare che non ci sono divisori > 1, ma ciò richiederebbe diverse vite dell'universo!
RSA utilizza un test di primalità probabilistico, cioè non si può affermare con certezza che l'esito del test sia corretto. Tuttavia, la probabilità di errore può essere resa arbitrariamente piccola aumentando il tempo di test. Il test si basa sul teorema di Fermat che fornisce una condizione necessaria affinché un intero n sia primo:

- se n è primo $\Rightarrow$ per ogni intero a risulta $a^{n-1} = 1\,mod\,n$;
- non vale però il viceversa: esistono degli interi a per i quali, l'uguaglianza è verificata anche se n è non primo

Test di primalità probabilistico. Dato un intero n, un possibile test di primalità può consistere dei seguenti passi:

1. scegliere un intero $a < n$;
2. calcolare $a^{n-1}\,mod\,n$;

3. a. se il risultato è diverso da 1 $\Rightarrow n$ è certamente non primo.
   b. se il risultato è pari a 1 $\Rightarrow n$ potrebbe essere primo, anche se non è sicuro(è stato dimostrato che, se n è un intero random di circa cento cifre decimali, la probabilità di un falso positivo è 10^{-13}).

Si osservi che un errore nel test di primalità può rendere impossibile la decifratura RSA di un messaggio, più facile l'identificazione della chiave privata.
Se una probabilità di errore pari a 10^{-13} non è ritenuta sufficiente, si possono effettuare più test con diversi valori di a: si ha che la $Pr\{falso\,positivo\,dopo\,k\,test\} = (10^{-13})^k$. La probabilità di errore può essere resa arbitrariamente piccola, ma non sempre è facile! Infatti, ci possono essere dei casi veramente sfortunati non rilevabili dal test, ad esempio se n è un numero di Carmichael: un numero n è detto di Carmichael se non è primo e se per ogni $a \leq n$ risulta $a^{n-1} = 1 \, mod \, n$. Tuttavia, i numeri di Carmichael sono sufficientemente rari che è estremamente improbabile estrarli a caso.

Calcolo di d ed e. Gli interi d ed e sono definiti nel seguente modo:

- e è un qualunque numero relativamente primo rispetto all'intero $\phi(n) = (p-1)(q-1)$;
- d è l'intero tale che $ed \, mod \, \phi(n) = 1 \Rightarrow$ noto e, d si calcola con l'algoritmo di Euclide.

Esistono due strategie per il calcolo di e:

1. una volta ottenuti p e q, si sceglie randomicamente e e si testa se esso è relativamente primo con $(p-1)(q-1)$; in caso negativo si ritenta con un altro valore di e.
2. Non selezionare prima p e q, al contrario, si sceglie prima e, per poi scegliere p e q tali che la quantità $(p-1)(q-1)$ sia relativamente prima con e.

La sicurezza di RSA non viene messa in crisi se e è scelto sempre allo stesso modo: d continua ad essere imprevedibile se p e q non sono noti. Se e è un intero piccolo o facile da calcolare, le operazioni di cifratura e di verifica della firma diventano più efficienti, cioè le operazioni che richiedono l'uso della chiave pubblica $PU =< e, n >$ sono più veloci, mentre risulta invariata l'efficienza delle operazioni che richiedono la chiave privata $PR =< d, n >$. Chiaramente, diversamente da e, non si può assegnare a d un valore piccolo, sebbene ciò renderebbe molto più veloci le operazioni che usano la chiave privata PR. Infatti, la sicurezza di RSA verrebbe meno: così si renderebbe l'informazione vulnerabile ad attacchi a forza bruta, poichè d è l'esponente privato, a differenza di e che è esponente pubblico.
Di solito si usa $e = 3$, poichè è comodo lavorare con esponenti piccoli, in modo che il calcolo di $m^e \, mod \, n$ non sia computazionalmente costoso(il calcolo di $m^3 \, mod \, n$ richiede soltanto due moltiplicazioni) e la cifratura sia efficiente(così come la verifica della firma). Non si può scegliere $e = 2$, in quanto non è relativamente primo con $(p-1)(q-1)$, che è un numero pari.
La scelta $e = 3$ comporta alcune vulnerabilità:

- se il messaggio m da cifrare rappresenta un intero piccolo, in particolare se $m < n^{1/3} \Rightarrow c = m^e \, mod \, n = m^3 \, mod \, n = m^3 \Rightarrow$ un avversario può decifrare c senza conoscere la chiave privata semplicemente estraendo la radice cubica ordinaria di $c \Rightarrow m = c^{1/3}$. Tale vulnerabilità può essere rimossa eseguendo un padding random del messaggio tale che $m^3 > n$. Ciò garantisce che m^3 viene sempre ridotto $mod \, n$.
- se uno stesso messaggio m viene inviato cifrato a tre o più destinatari aventi un esponente pubblico $e = 3$, il messaggio in chiaro m può essere decifrato conoscendo soltanto i tre messaggi cifrati c_1, c_2 e c_3 e le tre chiavi pubbliche $< 3, n_1 >$, $< 3, n_2 >$ e $< 3, n_3 >$: si supponga infatti che un avversario intercetti tre cifrature dello stesso messaggio m, cioè c_1, c_2 e c_3. Conoscendo anche le tre chiavi pubbliche $< 3, n_1 >$, $< 3, n_2 >$ e $< 3, n_3 >$ e utilizzando il teorema cinese del resto, l'avversario può calcolare $m^3 \, mod \, n_1 \, n_2 \, n_3$. Essendo $m < n_i$, per $i = 1, 2, 3 \Rightarrow m^3 < n_1, n_2, n_3$, da cui si ricava che $m^3 \, mod \, n_1 \, n_2 \, n_3 = m^3 \Rightarrow$ l'avversario può risalire ad m estraendo una radice cubica ordinaria. Anche questa vulnerabilità può essere rimossa mediante un padding random, così si evita che uno stesso messaggio cifrato venga inviato a piu destinatari.

Si osservi che nelle applicazioni pratiche di RSA,il messaggio m è generalmente una chiave di un algoritmo di cifratura a chiave segreta e in ogni caso m è molto più piccolo di n, per cui è sempre possibile aggiungere dei bit di riempimento(padding) in modo tale che il messaggio risultante presenti delle caratteristiche desiderate. Se per ogni destinatario il padding scelto è random, la precedente vulnerabilità viene rimossa; la vulnerabilità può essere rimossa anche usando come padding gli identificatori univoci (ID) dei destinatari.
Un'altra scelta possibile è $e = 65537$. Infatti esso è pari a $2^{16} + 1$, che è un numero primo, e rimuove o riduce del tutto le vulnerabilità viste nel caso $e = 3$: la prima vulnerabilità con $e = 3$ si ha se $m^3 < n$ e nel caso $e = 65537$ non ci sono molti valori di m tali che $m^{65537} < n$, a meno che n non sia molto più lungo di 512 bit, quindi l'estrazione della 65537-esima radice ordinaria di m non costituisce una vulnerabilità seria; la seconda vulnerabilità con $e = 3$ si ha se uno stesso messaggio m cifrato è inviato a 3 destinatari e nel caso $e = 65537$, lo stesso tipo di vulnerabilità si ha quando m viene inviato a 65537 destinatari e non si può dire certo che si tratti di un messaggio segreto!.
Infine, la scelta di fissare a priori $e = 3$ ha richiesto di scegliere n in modo tale che $\phi(n)$ e 3 fossero relativamente primi. Nel caso $e = 65537$ conviene generare p e q come se e non fosse prefissato e rigettare ogni valore di p o q che è uguale a $1 \, mod \, 65537$. Tale evento si verifica con una probabilità molto piccola, cioè 2^{-16}.
Sono presenti altri tipi di vulnerabilità: nel caso della firma digitale risulta che, per ogni numero $x < n$, x è la firma digitale del messaggio $m_x = x^e \, mod \, n$, infatti, $m_x^d \, mod \, n = (x^e \, mod \, n)^d \, mod \, n = x^{ed} \, mod \, n = x^{1 \, mod \, \phi(n)} \, mod \, n = x \, mod \, n = x \Rightarrow$ è banale falsificare la firma di qualcuno se il messaggio m da firmare non interessa. La difficoltà sta però nel falsificare la firma di uno specifico messaggio.
Generalmente, ciò che viene firmato(messaggio + padding) ha una struttura sufficientemente vincolata: vengono inseriti dei bit di riempimento organizzati in pattern regolari;la probabilità che un numero random costituisca un messaggio (padding incluso) valido è trascurabile, cioè è estremamente improbabile che un numero random contenga i pattern regolari di bit. Tuttavia, visto che i numeri in RSA sono molto grandi un avversario ha a disposizione molti tentativi, dunque i pattern di riempimento vanno scelti in modo opportuno.
Si fa utilizzo dunque degli *smooth numbers*. Intuitivamente, uno smooth number è un numero scomponibile nel prodotto di (molti) numeri primi ragionevolmente piccoli(non conviene usare una definizione assoluta, ovvero un numero è piccolo o grande in base alle capacità di calcolo dell'avversario). Ad esempio, il numero 6056820 è più smooth del numero 6567587, poiché $6056820 = 22 \cdot 32 \cdot 5 \cdot 7 \cdot 11 \cdot 19 \cdot 23$, mentre $6567587 = 13 \cdot 557 \cdot 907$. Si tratta di una vulnerabilità prevalentemente teorica, nella pratica difficilmente realizzabile, poichè richiede un'enorme capacità di calcolo, la raccolta di un numero elevato di messaggi firmati e molta fortuna(per l'avversario).

Idea base: dalle firme s_1 e s_2 dei messaggi m_1 ed m_2, è possibile calcolare le firme dei messaggi $m_1 \cdot m_2$, m_1/m_2, m_1^j, m_2^k e $m_1^j \cdot m_2^k$. Ad esempio, conoscendo la firma $s_1 = m_1^d \, mod \, n$, è possibile ottenere la firma di m_1^2 senza conoscere d(chiave privata): infatti, $(m_1^2)^d \, mod \, n = (m_1^d)^2 \, mod \, n = (m_1^d \, mod \, n)^2 \, mod \, n$, ottenendo quindi la firma di m_1^2. Se un avversario riesce a collezionare molti messaggi firmati, può ottenere la firma di ogni messaggio m esprimibile come prodotto e/o divisione di messaggi della collezione. In particolare, se ottiene le firme di due messaggi m_1 e m_2 tali che il rapporto $m_1/m_2 = p$, dove p è un numero primo, l'attaccante può calcolare la firma di p. Inoltre, se è abbastanza fortunato da raccogliere molte coppie di questo tipo, egli può calcolare la firma di molti numeri primi, quindi può falsificare la firma di ogni messaggio dato dal prodotto di ogni sottoinsieme di tali numeri primi ciascuno elevato ad una qualunque potenza. Con abbastanza coppie, può falsificare la firma di ogni messaggio rappresentato da uno smooth number.
Generalmente, ciò che si firma con RSA è un digest messaggio con padding $m^* = pad(h(m))$. Al digest del messaggio m vengono aggiunti, in modo opportuno, dei bit di riempimento(padding) ottenendo m^*. Se i bit di riempimento sono degli zeri, anzichè essere random, è piu probabile che m^* sia uno smooth number. Invece, è estremamente improbabile che un numero random $mod \, n$ sia smooth:

- con un padding a sinistra di soli zeri, l'intero da firmare $m_p = h(m)$ rimane piccolo, per cui il padding non riduce la probabilità che m_p sia smooth;
- con un padding a destra di soli zeri, $m_p = h(m) \cdot 2^k$ è un intero molto più grande, ma è divisibile per una potenza di due, quindi, analogamente, il padding non riduce la probabilità che m_p sia smooth;
- con un padding a destra random, l'intero da firmare m_p è estremamente improbabile che sia smooth.

Tuttavia, si espone RSA alla minaccia nota come il problema della radice cubica: si assuma che si è optato per padding a destra random per ridurre la probabilità che le firme prodotte siano smooth. Si ha l'inconveniente che, se l'esponente pubblico $e = 3$, allora un attaccante può virtualmente falsificare la firma di un qualsiasi messaggio. Infatti, supponiamo che un attaccante, Carol, voglia falsificare la firma di un qualche messaggio m avente digest h_m. Allora Carol applica un padding a destra di h_m, considerando bit a zero e ottenendo $p_m = h_m 00..00$. Poi calcola la radice cubica ordinaria e la arrotonda all'intero più vicino $r = round(p_m^{1/3})$, ottenendo la firma falsificata di m(infatti, $r^e = r^3 = p_m$, ossia h_m con un padding a destra che è apparentemente casuale).

5.3 PKCS

Ogni applicazione di RSA, cifratura, decifratura e firma, può essere soggetta a diversi tipi di attacchi, che possono essere sventati con opportune contromisure, basate sulla scelta di un'opportuna codifica/formato(quindi padding) del messaggio da cifrare/firmare. A tal fine è stato definito uno standard, **PKCS**(Public-Key Cryptography Standard), che stabilisce le codifiche per la chiave pubblica RSA, chiave privata RSA, firma RSA, cifratura RSA di messaggi corti(cioè chiavi segrete), firma RSA di messaggi corti(tipicamente digest).
Esistono 15 standard PKCS per le diverse situazioni in cui la cifratura a chiave pubblica viene utilizzata. Noi esamineremo solo PKCS1. Esso è stato concepito per far fronte alle seguenti minacce:

- cifratura di messaggi prevedibili;
- smooth number per le firme;
- destinatari multipli di un messaggio quando $e = 3$;
- cifratura di messaggi di lunghezza inferiore ad un terzo della lunghezza di n quando $e = 3$;
- firma di messaggi dove l'informazione è posta nei bit più significativi ed $e = 3$.

PKCS definisce uno standard per la formattazione di un messaggio da cifrare con RSA.
PKCS definisce uno standard per la formattazione di un messaggio da firmare con RSA.

5.4 Diffie-Hellman

Diffie-Hellman è il primo sistema a chiave pubblica utilizzato. Meno generale di RSA, non serve né a cifrare/decifrare né a firmare messaggi ma permette lo scambio di chiavi, in chiaro e su una rete pubblica insicura, tra due entità(che chiameremo Alice e Bob) e quindi di accordarsi su un segreto(chiave) condiviso, senza rivelarlo. Intercettando tutti i messaggi scambiati non si è in grado di risalire al segreto condiviso. Tale segreto non viene generato da una delle due entità, ma è il risultato dello scambio dei messaggi. In particolare, dopo essersi scambiati complessivamente due messaggi(in chiaro), che tutto il mondo può conoscere, Alice e Bob conosceranno il segreto condiviso K_{AB}, il quale verrà poi usato per proteggere la confidenzialità con tecniche di cifratura convenzionali.
Diffie-Hellman è realmente usato per stabilire una chiave segreta condivisa in alcune applicazioni, ad esempio nell'ambito della cifratura dei dati inviati in una LAN. Si osservi comunque che

Diffie-Hellman non incorpora alcuna forma di autenticazione, senza la quale si rischia di condividere il segreto con un impostore.

Algoritmo:

PRECONDIZIONE:

- Alice e Bob condividono due numeri p e g, dove p è un numero primo grande e $g < p$ con g radice primitiva di p
- p e g sono noti in anticipo e possono essere resi di dominio pubblico in una repository accessibile sia da Alice che da Bob, oppure possono essere generati dall'iniziatore della comunicazione, diciamo Alice, e trasmessi a Bob nel messaggio che gli invierà;

La fase 0, viene eseguita dall'iniziatore della comunicazione qualora i numeri p e g non siano pubblici: Alice genera (o estrae da un suo archivio) una coppia di numeri p e g tali da soddisfare le proprietà sopra elencate; p e g possono essere subito inviati a Bob oppure possono essere trasmessi nella fase 3a.

1. a. Generazione del segreto privato s_A: Alice(iniziatore comunicazione) genera un numero random $s_A < p$ di 512 bit, che non verrà mai inviato a Bob, quindi calcola $T_A = g^(s_A) \, mod \, p$ e lo invia(su una rete insicura) a Bob;
   b. Generazione del segreto privato s_B: Bob genera $s_B < p$ di 512 bit, che non verrà mai inviato ad Alice, quindi calcola $T_B = g^{s_B} \, mod \, p$ e lo invia in rete ad Alice;
2. a. Alice calcola $K_{AB} = T_B^{s_A} \, mod \, p$;
   b. Bob calcola $K_{BA} = T_B^{s_B} \, mod \, p$;
3. L'aritmetica modulare garantisce che $K_{AB} = K_{BA}$. Infatti si ha $K_{AB} = T_B^{s_A} \, mod \, p = (g^{s_B} \, mod \, p)^{s_A} \, mod \, p = g^{s_B s_A} \, mod \, p = g^{s_A s_B} \, mod \, p = (g^{s_A} \, mod \, p)^{s_B} \, mod \, p = T_A^{s_B} \, mod \, p = K_{BA}$.

Tuttavia, dati p, g, T_A e T_B, è possibile calcolare s_A, s_B oppure $g^{s_A s_B}$? Si può ottenere s_A da $g^(s_A)$ tramite logaritmo discreto $dlog_g \, g^{s_A}$, ma al momento non sono note tecniche per calcolare tale quantità in un tempo ragionevole, anche conoscendo g^{s_A} e g^{s_B}.
La vulnerabilità di Diffie-Hellman è che tale algoritmo non fornisce alcuna prova di autenticazione: Alice non può essere certa che T_B sia stato inviato da Bob e non da un impostore; allo stesso modo, Bob non può essere certa che T_A sia stato inviato da Alice e non da un impostore. E' quindi possibile che un impostore, Mr. X, intercetti e modifichi i messaggi facendo credere a Bob di comunicare con Alice e viceversa. Alice e Bob non hanno modo di rendersi conto dell'attacco in atto. Un attacco di questo tipo viene detto Man-in-the-Middle(o Bucket Brigade Attack): Alice pensa che K_{AX} sia la chiave segreta K_{AB} che condivide con Bob; Bob pensa che K_{bX} sia la chiave segreta K_{BA} che condivide con Alice; invece, Mr. X ha due chiavi segrete:

- K_{XA} per comunicare con Alice;
- K_{XB} per comunicare con Bob.

Questo perchè Mr. X conosce g e p, quindi si calcola $s_x < p$ e $T_x = g^{s_x} \, mod \, p$ e li invia sia ad Alice che a Bob. Tale attacco è un attacco all'integrità.
Come risolvere tale problema?

- **Autenticazione via password**:
  Ipotesi: Alice e Bob si sono preliminarmente accordate su una coppia di password(pwd_A, password che Alice invia a Bob; pwd_B, password che Bob invia ad Alice). Si consideri allora la seguente procedura di autenticazione, dove K_{AB} è la chiave segreta condivisa ottenuta con Diffie-Hellman:

  1. scambio chiavi Diffie-Hellman;

2. Alice invia un messaggio cifrato con K_{AB} e pwd_A;
3. Bob invia un messaggio cifrato con K_{BA} e pwd_B.

Il problema è che l'autenticazione via password, se è in corso un attacco Man-in-the-Middle, non funziona: Mr. X può decifrare tutte i messaggi che riceve da Alice con K_{AX}, cifrarli con K_{XB} e inviarli a Bob. Viceversa, può decifrare tutte i messaggi che riceve da Bob con K_{XB}, cifrarli con K_{AX} e inviarli ad Alice.

- Anche altri tipi di proposte sono insufficienti(timestamp, domande personali..). Il problema è che Diffie-Hellman effettua delle operazioni invertibili(cifratura/decifratura) ed è sicuro solo nel caso di attacchi passivi(un intruso intercetta i messaggi, ma non li modifica). E' necessario proteggere l'integrità, per cui si adottano due strategie generali: Diffie-Hellman con Numeri Pubblici e Scambio Diffie-Hellman Autenticato.

Diffie-Hellman con Numeri Pubblici. Un possibile modo per sventare attacchi attivi è evitare che p, g, s_A, s_B, T_A e T_B vengano generati/calcolati ad ogni scambio. p, g, T_A e T_B potrebbero essere resi pubblici in una repository fidata, con p e g uguali per tutti gli utenti, mentre ogni utente U pubblica il proprio valore T_U, mantenendo privato il segreto s_U. Ne consegue che, se un avversario non è in grado di accedere alla repository e di modificare i valori pubblici, allora Diffie-Hellman diventa sicuro anche nel caso di attacchi attivi. Inoltre, non è più necessario lo scambio dei valori T_A e T_B: consultando la repository ogni utente A può ottenere la chiave K_{AB} che condividerebbe con l'utente B.

Scambio Diffie-Hellman Autenticato. Alice e Bob conoscono un qualche tipo di informazione che permette loro di autenticarsi reciprocamente, ovvero una chiave segreta condivisa K^{AB}, da non confondere con la chiave concordata con Diffie-Hellman K_{AB}, la propria coppia <chiave privata, chiave pubblica> e la chiave pubblica dell'altro. Si possono usare tale(i) informazione(i) per provare che sono realmente loro, e non un impostore, coloro che generano i valori di Diffie-Hellman g, p, T_A e T_B. Tale prova può avvenire sia contestualmente che dopo lo scambio Diffie-Hellman esaminato in precedenza.
Alcune possibili soluzioni sono:

- Autenticazione contestuale allo scambio Diffie-Hellman:
  1. Cifrare lo scambio Diffie-Hellman con la chiave segreta K^{AB}.
  2. Cifrare il valore Diffie-Hellman con la chiave pubblica dell'altro interlocutore.
  3. Firmare il valore Diffie-Hellman con la propria chiave privata
- Autenticazione successiva allo scambio Diffie-Hellman:
  1. Dopo lo scambio Diffie-Hellman, trasmettere un hash della chiave concordata K_{AB}, del proprio nome e della chiave segreta K^{AB}.
  2. Dopo lo scambio Diffie-Hellman, trasmettere un hash del valore Diffie-Hellman trasmesso e della chiave segreta K^{AB}.

Notazione adottata:

- K^{AB}: chiave segreta condivisa tra Alice e Bob prima di effettuare lo scambio Diffie-Hellman.
- K_{AB}: chiave concordata con Diffie-Hellman.
- K^{AB} {msg} : cifratura di msg con la chiave segreta K^{AB}, cioè $E(K^{AB}, msg)$.
- $\{msg\}_{Bob}$: cifratura di msg con la chiave pubblica di Bob, cioè $E(PU_{Bob}, msg)$.
- $[msg]_{Bob}$: firma di msg con la chiave privata di Bob, cioè $E(PR_{Bob}, msg)$.

Oltre alla mancanza di autenticazione, Diffie-Hellman classico presenta anche lo svantaggio che, la comunicazione cifrata con la chiave concordata, può avvenire soltanto dopo l'esecuzione di uno scambio attivo. In pratica, Alice non può inviare un messaggio cifrato a Bob prima di ricevere T_B. Tale problema puo essere ovviato introducendo le chiavi pubbliche Diffie-Hellman: una chiave pubblica D-H è una tripla $< p, g, T >$, dove $T = g^s \, mod \, p$, dove s è la corrispondente chiave privata. Le chiavi pubbliche vanno custodite in un luogo fidato e accessibile da tutti in modo sicuro. La chiave pubblica di Bob è $< p_B, g_B, T_B >$.

Di seguito mostriamo un esempio di procedura che consenta ad Alice di inviare un messaggio cifrato a Bob, con la chiave concordata K_{AB}, anche se Bob risulta essere inattivo, cioè Alice deve essere in grado di cifrare senza dover attendere alcuna risposta da Bob, il quale, una volta attivo, dovrà poter calcolare K_{AB} e decifrare il messaggio:

1. Alice genera $s_A < p_B$, calcola $T_A^* = g_B^{s_A} \, mod \, p_B$ e $K_{AB} = T_B^{S_A} \, mod \, p_B$.
2. Poi cifra il messaggio $msg = E(K_{AB}, msg)$ e lo invia a Bob assieme a T_A^*.
3. Bob, una volta attivo, calcola $K_{BA} = (T_A^*)^{S_B} \, mod \, p_B$.
4. Decifra $E(K_{AB}, msg)$, ottenendo $msg = D(K_{BA}, E(K_{AB}, msg))$.

5.5 Zero Knowledge Proof System

La dimostrazione della conoscenza di un segreto è alla base di molte tecniche di autenticazione. Nelle tecniche di autenticazione a chiave segreta, il segreto è appunto la chiave condivisa tra i due principal. Nei protocolli a chiave pubblica il segreto è noto solo ad un principal, il quale deve dimostrare all'altro che detiene il segreto senza fornire delle informazioni che possano consentire ad un impostore di eseguire la prova.
Una dimostrazione quindi è a *conoscenza zero*(**Zero Knowledge Proof, ZKP**) se permette di provare la conoscenza di un segreto, che deve essere associato alla chiave pubblica, senza fornire delle informazioni che permettano ad un impostore di eseguire la prova:

- la prova non deve rivelare il segreto;
- la prova non deve rivelare eventuali informazioni, che pur non essendo il segreto, consentano comunque ad un impostore di effettuare la prova.

Le dimostrazioni a conoscenza zero sono impiegate nei protocolli/sistemi di autenticazione(detti **Zero Knowledge Proof Systems, ZKPS**). RSA è un esempio di ZKPS: è possibile provare la conoscenza di un segreto associato alla chiave pubblica(si pensi alla firma di una sfida), senza rivelare la chiave privata o altre informazioni che permettano ad un impostore di impersonare il proprietario della chiave privata. Tuttavia, esistono ZKPS molto più efficienti di RSA, anche se non permettono di cifrare e/o di firmare.

Uno schema di autenticazione a conoscenza zero(**Zero Knowledge Authenication Scheme, ZKAS**) consiste in un'autenticazione che sfrutta una *ZKP*. Non si tratta di una tecnica deterministica, ma probabilistica(anche RSA in fondo è probabilistica). Deve poter essere resa arbitrariamente piccola la probabilità che un dimostratore onesto fornisca una prova errata; un verificatore onesto fornisca una verifica errata quando il dimostratore è onesto; un dimostratore disonesto fornisca una prova corretta.
Uno schema di autenticazione a conoscenza zero deve soddisfare i seguenti requisiti:

a. ad ogni entità è associato un segreto privato s e una chiave pubblica k_s, cioè una coppia $< s, k_s >$. Ovviamente, k_s non deve esporre s;
b. l'autenticazione consiste nel provare la conoscenza del segreto s;

c. la prova deve essere a conoscenza zero, cioè le informazioni addotte dal dimostratore non devono poter essere riutilizzate con successo(in seguito) da un impostore, quindi la prova non consente di rivelare s;

d. chi non conosce il segreto s non deve poter eseguire la prova con successo;

e. chi non conosce il segreto s deve poter verificare la correttezza della prova utilizzando la chiave pubblica k_s dell'entità che si sta autenticando(senza la chiave pubblica non deve essere possibile verificare la correttezza della prova).

5.5.1 ZKAS basato su MSR

Di seguito mostriamo un protocollo di autenticazione, estremamente efficiente, che sfrutta un problema difficile nell'ambito dell'aritmetica modulare. A rigore tale schema di autenticazione non è completamente a conoscenza zero, anche se nella pratica può considerarsi tale.
Il problema che viene sfruttato è il problema della radice quadrata modulare(**Modular Square Root, MSR**): dati un numero intero semiprimo grande $n = pq$, con p e q numeri primi grandi, $m < n$ intero assegnato(avente una radice quadrata ordinaria non intera), trovare un numero intero s tale che $s^2 \, mod \, n = m$ è un problema difficile almeno quanto fattorizzare un numero intero. Dunque, tale protocollo consiste dei seguenti passi:

1. **Generazione delle chiavi**. Peggy(il dimostratore) calcola la chiave pubblica $< n, v >$, dove $n = pq$ come in RSA, v è un numero di cui Peggy conosce la radice quadrata modulare(ottenere v è semplice, basta scegliere un numero random s e porre $v = s^2 \, mod \, n$; s è la chiave privata di Peggy da non rivelare; $< n, v >$ va divulgata a tutto il mondo.

2. **Autenticazione**.

   - Peggy sceglie k numeri random $r_1, r_2, .., r_k$;
   - Per ogni r_i invia a Victor $r_i^2 \, mod \, n$;
   - Victor attribuisce a ciascun r_i^2 che vale 0 o 1 e la comunica a Peggy;
   - Peggy invia a Victor $sr_i \, mod \, n$ per ciascun r_i^2 etichettato con 1 e $r_i \, mod \, n$ per ciascun r_i^2 etichettato con 0;
   - Victor eleva al quadrato ciascun numero della risposta di Peggy e verifica che tale quadrato valga $vr_i^2 \, mod \, n$ se il corrispondente r_i^2 aveva etichetta 1, oppure $r_i^2 \, mod \, n$ se il corrispondente r_i^2 aveva etichetta 0.

Supponiamo che Fred voglia impersonare Peggy. Allora egli è in grado di rispondere in modo corretto agli r_i^2 che Victor etichetta con 0(Fred può scegliere a suo piacimento gli r_i), ma non è in grado di rispondere agli r_i^2 etichettati con 1. In assenza di etichette 0, infatti, il protocollo si semplificherebbe: Peggy si limiterebbe ad inviare delle coppie $< r_i^2, sr_i \, mod \, n >$, tuttavia non si avrebbe più un protocollo a conoscenza zero, poichè Fred potrebbe usare una precedente sequenza inviata di Peggy ed impersonarla con successo. Invece, l'etichettatura scelta in modo casuale da Victor implica che Fred ha una probabilità del 50% di rispondere in modo corretto ad ogni r_i^2. Fred potrebbe generarsi autonomamente gli r_i, ma in tal caso non saprebbe rispondere agli r_i^2 con etichetta 1 oppure potrebbe usare un insieme di r_i^2 etichettati in passato con 1 in una precedente autenticazione, ma allora non conoscerebbe i corrispondenti r_i e non saprebbe rispondere nel caso in cui l'etichetta è 0. Se k è sufficientemente grande la probabilità che Fred impersoni correttamente Peggy tende a 0.
ZKAS basato su MSR è molto più efficiente di RSA. Infatti, assumendo $k = 30$, Peggy deve effettuare 45 operazioni modulari(30 quadrati più una media di 15 moltiplicazioni per s) e Victor deve effettuare lo stesso numero di operazioni di Peggy; usando RSA Peggy deve eseguire una esponenziazione modulare che consiste in una media di 768 moltiplicazioni modulari mentre Victor se la cava con 3 moltiplicazioni nel caso in cui $e = 3$.

Capitolo 6

Sicurezza dei Sistemi Operativi

Un SO (Sistema Operativo), o OS (Operating System), gestisce il modo in cui le applicazioni software accedono alle risorse hardware del calcolatore:

- CPU
- Memoria principale
- Memoria secondaria
- Periferiche di I/O
- Interfacce di rete

Un OS fornisce un'interfaccia semplificata e consistente a utenti e applicazioni al fine di interagire con i componenti hardware. Grazie a questa astrazione è possibile sviluppare programmi software senza preoccuparsi della particolare tipologia di hardware sul quale saranno eseguiti. Grazie a questa astrazione è possibile sviluppare programmi software senza preoccuparsi della particolare tipologia di hardware sul quale saranno eseguiti. Gli OS svolgono numerose funzioni, alcune delle quali strettamente legate a problemi di sicurezza, in particolare vedremo:

- meccanismi di autenticazione
- sicurezza dei processi
- sicurezza del filesystem
- sicurezza della memoria

6.1 Meccanismi di Autenticazione

Un OS deve poter identificare i propri utenti in modo sicuro, i.e. utenti diversi potrebbero avere permessi di accesso alle risorse diversi. Un meccanismo di autenticazione standard ampiamente usato consiste nell'inserimento di un username e di una password: se la password inserita coincide con la password memorizzata dall'OS per il dato username allora l'utente viene autenticato. Un OS deve dunque memorizzare la password di ogni utente che può accedere al sistema. Generalmente gli OS memorizzano le password criptate attraverso funzioni hash in un file o in un apposito database. Grazie alla proprietà one-way delle funzioni hash, un attaccante che riesce ad accedere al file dove sono memorizzate le password non può ricostruire facilmente il loro valore. Tali file si trovano:

- per Windows in system32 - config - SAM
- per Linux in */etc/passwd* e in */etc/shadow* (solo l'utente root può leggerle)

6.1.1 Possibili attacchi

- **Brute Force**: attacco offline, tutte le possibili password per un dato alfabeto vengono generate automaticamente, criptate con la funzione hash usata dal sistema di autenticazione e confrontate con le password memorizzate
- **Dizionario**: attacco offline, liste di parole comuni (es: nomi) che vengono criptate con la funzione hash usata dal sistema di autenticazione e confrontate con le password memorizzate
- **Rainbow tables**

6.1.2 Password Robuste

Per la definizione di password robuste a tali attacchi vanno seguite delle ben definite linee guida:

- evitare parole comuni (e.g.: nomi)
- evitare password brevi
- usare caratteri maiuscoli E minuscoli
- usare caratteri speciali
- usare i numeri

Un esempio di password robusta è: "Voglio compr@re 11 Cani!": per scoprire questa password con un attacco brute-force in 60 giorni dovrei disporre di un computer in grado di generare circa $3,86x10^{44} PW/sec$. Per rendere la struttura ancora più sicura si può impostare una scadenza alla password.

6.1.3 Password Salt

Il **Salt** consiste nell'aggiunta di bit random all'input di una funzione hash (o di un algoritmo di crittografia) al fine di aumentare la randomicità dell'output. Nel caso dell'autenticazione è possibile associare un numero random all'userID dell'utente, seguendo la forma:

salt = numero random + userID

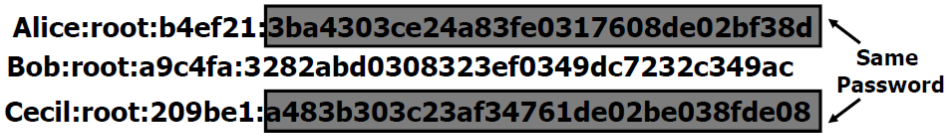


Figura 6.1: Esempi di Salt

Il processo di autenticazione funziona nel seguente modo:

- L'utente inserisce il suo userID X e la password P
- Il processo di autenticazione dell'OS recupera il salt S per l'userID X e l'hash H del salt concatenato alla password associata a X
- OS verifica se $HASH(S||P) == H$

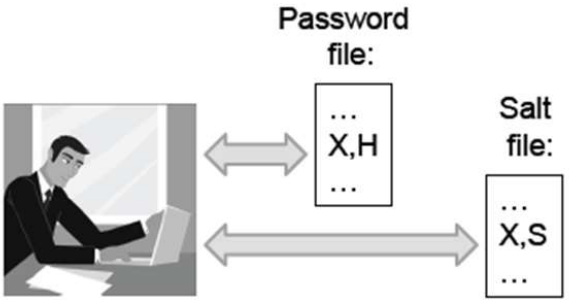


Figura 6.2: Processo di autenticazione con Salt

Tale meccanismo comporta evidenti benefici. Se l'attaccante non può trovare il salt associato con l'userID, allora lo spazio di ricerca per un attacco con dizionario cresce notevolmente, diventando $2^B xD$, dove B è il numero di bit del Salt, mentre D lo spazio del dizionario.
Inoltre, anche se l'attaccante fosse in grado di recuperare il salt memorizzato in forma criptata dall'OS, questo meccanismo consente di rallentare notevolmente l'attacco con dizionario, rendendolo valido per un userID alla volta. Senza salt è possibile crackare molte password nello stesso momento, in quanto si ha solo bisogno dell'hash di ogni possibile password e di confrontarlo con tutti gli hash memorizzati. Con il meccanismo di salt, invece, ogni possibile password va concatenata al salt ad essa associato prima di calcolarne l'hash.

Inoltre, è possibile che due utenti utilizzino la stessa password, o che lo stesso utente scelga di utilizzare la stessa password per due account diversi. Senza il meccanismo di salt le due password hanno lo stesso hash (questo potrebbe rivelare il fatto che i due account hanno la stessa password, permettendo a chiunque che conosce una delle due password di accedere anche all'altro account). Grazie al meccanismo di salt gli hash delle due password risultano invece diversi.

6.2 Sicurezza dei processi

Un processo è un'istanza di un programma in esecuzione: il codice di un programma in esecuzione viene caricato dalla memoria secondaria in cui è memorizzato e passato alla memoria primaria. Più istanze di uno stesso programma possono essere eseguite come processi diversi, e un'applicazione può essere composta da più di un processo. I processi attivi vengono eseguiti "parallelamente" attraverso la tecnica di time-sharing su ogni core della CPU. Ogni processo è univocamente identificato da un intero detto **PID** (**P**rocess **ID**) e viene associato all'user (utente) che lo ha generato.

Un processo può inoltre avviare e controllare altri processi. Un nuovo processo è generato mediante il meccanismo di **forking**. Il processo che richiede il forking è detto **parent process**, il processo forked è detto **child process**.

Listing 6.1: Esempio forking in C

```
int main()
{
        printf("I'm the parent, my PID is %d, my parent is process %d\n",
                        getpid(), getppid());
        fork();
        printf("This sentence has been printed by process: %d my parent
                        is process %d\n", getpid(), getppid());
}
```

6.2.1 Istruzione Fork

L'istruzione fork() crea una copia del processo corrente. Tale funzione ritorna 0 nel processo figlio e un intero maggiore di 0 nel processo padre (il valore restituito è proprio il PID del figlio). Ritorna invece (al padre) un intero minore di 0 nel caso in cui non sia stato possibile creare un nuovo processo. Al momento del fork() viene creato uno address space (porzione di memoria dedicata) separato per il processo figlio, il quale eredita una copia esatta di tutti i segmenti di memoria del processo padre (codice, stack, file descriptor, heap, variabili globali, e program counter).

Il meccanismo di forking porta ad un'organizzazione dei processi rappresentabile con una struttura ad albero radicato (**rooted tree**). In Linux a radice dell'albero è sempre il processo **init** che viene lanciato dal kernel durante il processo di boot del sistema operativo. Il processo init crea poi nuovi processi. In Windows e applicazioni lanciate dagli utenti sono sempre figlie del processo **Windows Explorer**.

6.2.2 Ulteriori nozioni sui processi - facoltativo

Comunicazione tra processi

Esistono diversi meccanismi per la comunicazione tra processi:

- lettura/scrittura di files: semplice ma inefficiente e poco sicuro
- porzione di memoria RAM condivisa. Questa soluzione è veloce e efficiente, ma il kernel deve gestire in maniera sicura la separazione tra porzioni di memoria condivise e private
- pipes e sockets: oggetti in RAM condivisi che fungono da canale virtuale tra due processi
- signals: notifiche asincrone, il processore interrompe il flusso del processo ricevente e verifica se esiste un gestore per la notifica (routine da eseguire)

Demoni e Servizi

- **Linux**: i **daemons** sono processi lanciati prima dell'autenticazione dell'utente, tipicamente dal processo init. Tali processi possiedono permessi più alti di qualsiasi utente, sopravvivono al termine delle sessioni utente e sono indistinguibili dagli altri processi
- **Windows**: esistono processi analoghi chiamati **services**. Rispetto ai daemons sono distinguibili dagli altri processi, poiché sono monitorati in modo specifico all'interno del TaskManger (esistono due tab separati, uno per i processi e uno per i servizi)

System Call

I processi comunicano con il kernel per inoltrare le richieste verso l'hardware. La comunicazione avviene mediante librerie dette **system call**. Ad ogni system call segue generalmente un interrupt

che vincola il processore a fermare l'esecuzione corrente e a gestire la chiamata. Possibili attacchi alla sicurezza degli OS prevedono la contraffazione di una system call al fine di eseguire codice malevolo ad ogni chiamata o danneggiamento di una system call al fine di compromettere il funzionamento del sistema. Sempre più spesso in realtà, i programmatori non usano direttamente le system call, ma delle API che fungono da strato intermedio tra le applicazioni e le system call, al fine di facilitare e migliorare la portabilità delle applicazioni. Le funzioni contenute in queste API invocano a loro volta opportune system call e spesso vi è una corrispondenza diretta tra una funzione API ed una corrispondente system call. Esistono tipi di System Call per ogni possibile operazione: controllo dei processi, gestione dei file, comunicazione fra processi, gestione dei dispositivi e gestione delle informazioni di sistema.

6.2.3 Processi e Utenti

Come già detto, ogni processo è associato ad un utente. Utenti specifici possono avere permessi maggiori rispetto agli utenti normali (e.g. installare o rimuovere programmi, modificare i permessi degli altri utenti, modificare la configurazione del sistema).

Nei sistemi Unix l'utente root non ha alcuna restrizione. Nei sistemi Windows esistono diversi utenti speciali: SYSTEM, LOCAL SERVICE e NETWORK SERVICE associati direttamente al sistema operativo, e uno o più utenti administrator. Laddove SYSTEM non ha restrizioni, administrator, LOCAL SERVICE e NETWORK SERVICE agiscono con permessi ridotti e specifici per i loro scopi.

Processi e Utenti - Linux

In Linux ad ogni processo sono associati 4 indici:

- un **uid** (user ID) che identifica l'utente che ha lanciato il processo
- un **gid** (group ID) che identifica il gruppo a cui appartiene l'utente –
- un **euid** (effective user ID) che può differire dall'uid e identificare l'utente proprietario del file eseguibile, il quale può avere permessi maggiori. L'euid prevale sull'uid solo nel caso in cui è settato il bit**setuid**
- un **egid** (effective group ID) che può differire dal gid e identificare il gruppo proprietario dell'applicazione, il quale può avere permessi maggiori. L'egid prevale sull'gid solo nel caso in cui è settato il bit **setgid**

Tramite questi id Linux è in grado di stabilire i permessi di un dato processo su una data risorsa. I permessi del processo figlio sono automaticamente ereditati dal processo padre. Molti program-

Listing 6.2: Esempio forking in C

```
chmod u+s <file_eseguibile> %attiva setuid
chmod u-s <file_eseguibile> %disattiva setuid
chmod g+s <file_eseguibile> %attiva setgid
chmod g-s <file_eseguibile> %disattiva setgid
```

mi che accedono risorse di sistema hanno il bit setuid settato e sono detti **setuid programs** (es: passwd, su). Questo meccanismo può essere soggetto ad attacchi di scalata dei privilegi, che vedremo più avanti.

Vediamo un esempio. Prendiamo un file eseguibile chiamato *app*, con *userA* e *groupA* UID e GID dell'utente proprietario del file e *userB* e *groupB* UID e GID dell'utente che lancia app,

producendo il processo *P*. Se il bit SETUID di app è attivo, l'EUID di P è *userA*, il EGID di P è *groupA*. Se il bit SETUID di app non è attivo, EUID e UID coincidono con *userB*, EGID e GID coincidono con *groupB*. Supponiamo ora che il processo *P* voglia accedere ad un file di nome *info*. Se EUID di P coincide con il proprietario di *info*, il processo acquisisce i diritti di accesso del proprietario di *info*. Altrimenti se EGID di P e il gruppo di *info* coincidono, P acquisisce i diritti di accesso del gruppo di utenti associato a *info*. Se nessuna delle due precedenti condizioni è valida, valgono i normali diritti di accesso che vedremo più avanti (lettura/scrittura/esecuzione), l'accesso sarà consentito o meno a seconda della categoria di utenti nella quale ricadono UID e GID del processo P.

6.3 Sicurezza del filesystem

Il filesystem è un altro componente fondamentale di un OS, in quanto fornisce un'astrazione sull'organizzazione della memoria secondaria del calcolatore. Gli OS organizzano tipicamente i file in una gerarchia: ogni cartella può contenere file e/o sottocartelle. In questo modo è possibile rappresentare la memoria secondaria attraverso una struttura ad albero radicato. Prima di affrontare il tema della sicurezza del filesystem, diamo due definizioni:

- **Principal**: utente o gruppo di utenti
- **Permission**: azione possibile (e.g.: read, write, execute, list)

Access Control Entities

Una **Access Control Entry, ACE,** è una terna $\langle principal, type, permission \rangle$, che serve a descrivere i permessi che gli utenti hanno sui file. Per una data risorsa ed un dato principal specifico chi può fare cosa attraverso entries di questo tipo, in cui il valore **type** può assumere **allow** o **deny**.

Una **Accesso Control List** è invece una lista ordinata di terne **ACE**. Una possibile gestione per la sicurezza del filesystem prevede l'associazione tra ogni file e directory e un ACL che definisce la politica di accesso alla risorsa.

Discretionary Access Control

Il **DAC (Discretionary Access Control)** è un modello di assegnazione dei permissi in cui il creatore (owner) di un file o directory ha il potere di modificare i permessi ad esso associati (ACE). Sono disponibili due tipi di politiche di assegnazione dei permessi:

- **Closed policy o default sicuro**: solo ACE di tipo allow, tutti i permessi non esplicitamente concessi sono automaticamente negati
- **Open policy**: ACE tipo sia ALLOW che DENY

6.3.1 Sicurezza del filesystem Linux

Per la gestione dei permessi Linux utilizza una politica **DAC** con **default sicuro**. Inoltre l'accesso a un file dipende dall'ACL del file e di tutte le directory antenate: partendo dalla directory root, tutte le directory antenate devono avere il permesso **execute** (che per una directory significa il permesso di accedere alla stessa) per l'utente specificato.

Linux utilizza una versione semplificata delle ACL: la **File permission matrix**, una matrice di permessi associata ad ogni file e directory (tale metodologia rappresenta uno standard per tutti i sistemi Unix-like). La matrice rappresenta tre principal: **owner**, **group** (utenti del gruppo a cui afferisce l'owner) e **others**. I permessi di ognuna delle 3 classi sono definiti da 3 bit:

- **read bit**: per i file rappresenta il permesso di lettura, mentre per le directory il permesso di lettura della lista dei file contenuti
- **write bit**: per i file rappresenta il permesso di scrittura, mentre per una directory il permesso di creazione e eliminazione dei file in essa contenuti
- **execute bit**: per i file rappresenta il permesso di esecuzione, mentre per le directory il permesso accesso (l'utente può cambiare la sua cartella corrente con quella in questione)

-rw-r—r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden to others
-rwx------	read/write/execute for owner, forbidden to everyone else
-r--r--r--	read-only to everyone, including owner
-rwxrwxrwx	read/write/execute to everyone

Figura 6.3: Esempi di permission matrix

Per visualizzare le permission matrix delle directory e dei file contenuti in una directory, basta accedere a tale cartella e utilizzare il comando **ls -l**.

Oltre i tipi di permessi esaminati finora, esistono altri tipi di permessi cosidetti **speciali**, i quali consentono di impostare determinate funzioni avanzate sui file.

- **Set-user-ID (suid o setuid) bit**: su file eseguibili, causa l'esecuzione del processo con i diritti dell'utente owner
- **Set-group-ID (sgid o setgid) bit**: su file eseguibili, causa l'esecuzione del processo con i diritti del gruppo del file
- **Sticky bit**: sulle directory, vieta a tutti gli utenti di eliminare o rinominare file di cui non sono i creatori

Il bit **setuid** risolve il problema dell'accesso senza privilegio. Ad esempio, se un utente volesse modificare la propria password attraverso il programma di sistema **passwd** non potrebbe farlo in quanto non avrebbe i permessi per accedere al file **etc/passwd**. In realtà **passwd** viene eseguito con i privilegi dell'utente root grazie al bit setuid.

Scalata dei privilegi

Se un attaccante riesce a forzare un **programma setuid** a eseguire codice arbitrario (ad esempio tramite un attacco di tipo buffer overflow) può compromettere il sistema, realizzando uno scenario di attacco chiamato **scalata dei privilegi (o privileges escalation)**. Questo meccanismo va gestito attraverso tecniche di programmazione sicura.

Notazione Ottale

Nei sistemi Linux i bit di permessi sono espressi in notazione ottale (con 3 o 4 cifre). Le cifre sono organizzate nel seguente modo:
[special bits][user bits][group bits][others bits]. Ogni sezione deve prevedere ovviamente tre bit: uno per la scrittura, uno per la lettura ed uno per l'esecuzione.

644 or 0644	read/write for owner, read-only for everyone else
775 or 0775	read/write/execute for owner and group, read/execute for others
640 or 0640	read/write for owner, read-only for group, forbidden to others
2775	same as 775, plus setgid (useful for directories)
777 or 0777	read/write/execute to everyone *(dangerous!)*
1777	same as 777, plus sticky bit

Figura 6.4: Esempi di permission matrix in notazione ottale

- per i bit user/group/others, i bit sono nell'ordine r-w-x. Una terna $(100)_2 = (4)_8$, ad esempio, significa permesso consentito in lettura, mentre una terna $(101)_5 = (5)_8$, invece, permesso consentito in lettura ed esecuzione, ma non in scrittura.
- per i speciali, i bit sono nell'ordine setuid - setgid - sticky. Quindi avrò $(100)_2 = (4)_8$ se setuid, $(010)_2 = (2)_8$ se setgid e $(001)_2 = (1)_8$ se sticky (e ovviamente le varie combinazioni possibili)

6.3.2 Sicurezza del filesystem Windows

Per la gestione dei permessi Windows utilizza una politica **DAC** con **open policy**. Se non c'è una regola per un particolare principal o permesso allora l'accesso è negato di default.A differenza di Linux, al fine di accedere ad un file o ad una directory, è sufficiente avere i permessi per l'azione richiesta sul dato file o directory. In questo modo è possibile negare l'accesso ad una data directory pur consentendo l'accesso alle directory figlie (assurdo).

Vi sono due tipi di permessi: espliciti ed ereditati. Il concetto di permessi ereditati deriva dalla possibilità di estendere ogni permesso applicato ad una directory alle directory figlie.
I permessi espliciti standard sono: **modify, read and execute, read, write full control**. E' possibile comporre più **permessi standard** creando **permessi avanzati**.

Per quanto riguarda il controllo dei permessi va notato che:

- Il permesso DENY ha precedenza sul permesso ALLOW
- I permessi espliciti hanno precedenza sui permessi ereditati
- I permessi ereditati hanno precedenza tra di loro in base alla distanza dalla risorsa

File Descriptors

L'assegnazione dei permessi a file o cartelle, in Windows, viene gestita tramite il meccanismo dei file descriptors. Un **file descriptor** o **file handle** è un identificatore per un file o per una cartella. I file descriptor sono memorizzati in una tabella che li mappa con il percorso del file a cui si riferiscono, chiamata **file descriptor table**. L'accesso ai file e cartelle tramite System Calls passa quindi per questi costrutti. Le operazioni possibili sono:

- **open**: ritorna un file handle
- **read/write/execute** file
- **close** file: invalida il file handle

Quando un processo (programma) necessita di accedere ad un file in lettura o scrittura: viene richiamata una **open system call**. Il kernel verifica se il processo possiede i permessi necessari per accedere al file con l'azione richiesta e, se la verifica è positiva, crea un nuovo file descriptor e una nuova entry nella file descriptor table, quindi ritorna il file descriptor al processo. Il processo può leggere/scrivere sul file richiamando le relative read/write system call, e il kernel (attraverso la file descriptor table) effettua le letture/scritture direttamente sul file originale. Al termine dell'elaborazione del file il processo dovrebbe richiamare una close system call per rimuovere il file descriptor.

File Descriptors Leaks

Quando un processo crea un processo figlio (forking), quest'ultimo eredita una copia di tutti i file descriptors aperti dal processo padre. L'OS verifica i permessi soltanto al momento della creazione del file descriptor: al momento della lettura/scrittura sul file, i controlli effettuati si basano sui permessi associati al file descriptor (ad esempio un processo può sfruttare un file descriptor in lettura solo per leggere il file e non per scrivere). Un possibile scenario pericoloso potrebbe presentarsi nel caso che un processo con alti privilegi apra un file descriptor per un file protetto, prima di chiudere il file descriptor crei un nuovo processo con privilegi minori, il quale tuttavia erediterebbe comunque il file descriptor (in quanto ancora aperto). In tal caso, infatti, il processo figlio potrebbe usare il file descriptor pur non avendo i permessi necessari per operare sul file protetto.

```
[caption={Esempio codice vulnerabile al File Descriptors Leaks]
#include <stdio.h>
#include <unistd.h>
int main(int argc, char * argv[ ])
{
        /* Open the password file for reading */
        FILE *passwords;
        passwords = fopen("/home/admin/passwords", "r");
        /* Read the passwords and do something useful */
        /* . . . */
        /* Fork and execute Joe's shell without closing the file */
        execl("/home/joe/shell", "shell", NULL);
}
```

Per rimuovere questa vulnerabilità la funzione **fclose()** va richiamata prima del comando .

6.4 Sicurezza della memoria

Un ulteriore servizio offerto dagli OS è l'organizzazione e l'allocazione della memoria primaria. Quando un processo viene lanciato l'OS alloca una regione di memoria detta **address space** del processo. La gestione della memoria è trasparente per il processo: il processo "vede" a sua

disposizione l'intera memoria. Generalmente un processo non può avere accesso all'address space di un altro processo, se non per risorse esplicitamente condivise.

Nei sistemi Unix-like il modello address space è diviso in 5 settori:

- **Text**: memorizza il codice macchina del programma
- **Data**: memorizza le variabili statiche del programma inizializzate prima dell'esecuzione
- **BSS (Block Started By Symbol)**: memorizza le variabili statiche del programma non inizializzate
- **Heap**: settore dinamico, memorizza i dati generati durante l'esecuzione del programma (es: oggetti Java/C++)
- **Stack**: memorizza una struttura dati a pila, che tiene traccia delle chiamate a metodi e routine con i relativi argomenti e punti di ritorno

In Figura **??** è mostrato il modello. Si noti che lo Stack cresce verso il basso. In tale modello ogni settore ha i suoi permessi: il settore text è read-only in quanto il codice macchina del programma in esecuzione non deve poter essere modificato, mentre gli altri settori necessitano di essere writable in quanto i loro dati devono poter essere modificati durante l'esecuzione.

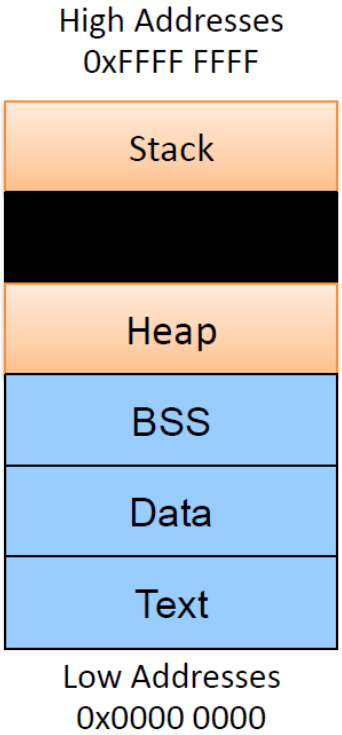


Figura 6.5: Modello Address Space Unix

6.5 Exploit

Un **exploit** è un input che sfrutta una vulnerabilità, un bug o un guasto di un sistema al fine di eseguire un attacco. Una metodologia tipica di attacco prevede:

- la ricerca di una vulnerabilità

- reverse engineering del codice
- costruzione dell'exploit

6.5.1 Buffer Overflow

Generalmente, con **buffer**, si intende una porzione di memoria contenente più istanze dello stesso tipo. In C e in molti altri programmi i buffer sono chiamati array. I buffer più comuni sono gli array di caratteri (stringhe). In C i buffer, come tutte le variabili, possono essere dichiarati statici o dinamici. Le variabili statiche sono allocate al momento del caricamento del programma sul segmento **data**, mentre le variabili dinamiche sono allocate in fase di esecuzione sullo **stack**.

Scenario

Un programma alloca in memoria un buffer (array) di dimensione fissata. Nel buffer viene poi copiato un input proveniente dall'utente, e la dimensione di tale input **non viene controllata** prima dell'operazione di copia.

Attacco

Un attaccante può confezionare un input malevolo di dimensione superiore a quella del buffer. Il programma copia l'input e sovrascrive regioni di memoria oltre al buffer. L'attaccante riesce a eseguire codice malevolo (**shellcode**) con i privilegi del programma attaccato. Il buffer riempito di codice malevolo viene denominato **payload**. L'attaccante solitamente inietta codice in grado di aprire un terminale (shell) attraverso cui eseguire altri comandi (e.g. Linux: /bin/sh, Windows: command.com). Ad esempio in un sistema Linux è possibile iniettare codice che richiama la funzione **setuid()** per poi aprire un terminale. Il codice viene generalmente iniettato direttamente sullo stack o sull'heap e pertanto deve essere scritto in codice operativo (**opcodes**) specifico per l'architettura della CPU attaccata.

6.5.2 Stack-Based Buffer Overflow

I moderni computer sono progettati tenendo in mente la necessità di poter usufruire di linguaggi di alto livello (con caratteristiche di **programmazione strutturata**, come C e C++). La tecnica più importante per la strutturazione dei programmi introdotta dai primi linguaggi di alto livello è la routine (function). Una chiamata a una procedura altera il flusso di controllo proprio come fa un salto, ma a differenza di un salto, al suo termine il controllo ritorna alla dichiarazione o istruzione successiva alla chiamata. Questo alto livello di astrazione è implementato con l'aiuto dello stack. Lo stack è infatti usato nell'ambito delle routine per allocare dinamicamente le variabili locali, per il passaggio dei parametri, e per la restituzione di valori.

Lo stack è un settore dell'address space contenente dati. La sua dimensione è regolata dinamicamente dal kernel in fase di esecuzione, tramite una politica LIFO e opportune chiamate a funzioni push/pop. Ogni chiamata ad una routine è associata ad un frame che memorizza le variabili locali, gli argomenti, e l'indirizzo di ritorno verso la chiamata padre. Alla base dello stack c'è quindi il frame relativo alla chiamata main(). Un buffer overflow che coinvolge una variabile locale può causare la sovrascrittura di parte della memoria allocata nello stack, con conseguenze pericolose.

Lo stack viene gestito attraverso opportune variabili e registri (vedi Figura **??**):

- ogni frame contiene un puntatore chiamato **Return address**, che punta all'indirizzo di memoria in cui la routine in questione è stata chiamata.

- l'ultima locazione di memoria occupata sullo stack (ovvero il top dello stack, poiché viene usata una politica LIFO) viene referenziata da un registro apposito, detto **Stack Pointer (ESP)**.
- la prima locazione di memoria del record di attivazione di una routine viene referenziata da un registro apposito, detto **Frame pointer (EBP)**.

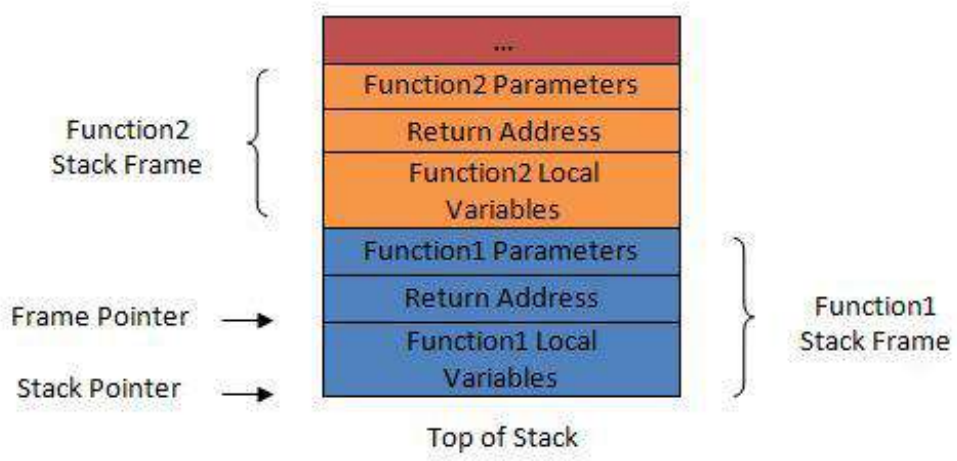


Figura 6.6: Stack Structure

Gli indirizzi delle variabili locali di una routine sono specificati rispetto al frame pointer poiché il suo valore rimane invariato per tutta la durata della routine stessa.

```
void function(int a, int b, int c)
{
   char buffer1[5];
   char buffer2[10];
}

void main()
{
   function(1,2,3);
}
```

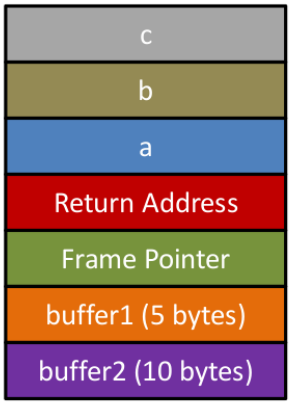


Figura 6.7: Esempio Stack di un programma in C

6.5.3 Esempio codice C

Prendiamo come esempio la funzione *strcpy(dest,src)*. Tale funzione (appartenente alle librerie C standard) copia la stringa *src* in *dest* senza verificare se la dimensione di src eccede quella di dest. Per rimediare a questa vulnerabilità basta utilizzare la funzione*strncpy(dest,src,n)*, che consente di specificare il numero di caratteri da copiare, e se la lunghezza di src supera tale soglia i caratteri in eccesso vengono scartati.

Listing 6.3: Esempio codice vulnerabile al buffer overflow in C

```
Main(int argc, char *argv[])
/* get user_input */
{
        char var1[15];
        char command[20];
        strcpy(command, "whois");
        strcat(command, argv[1]);
        strcpy(var1, argv[1]);
        printf(var1);
        system(command);
}
```

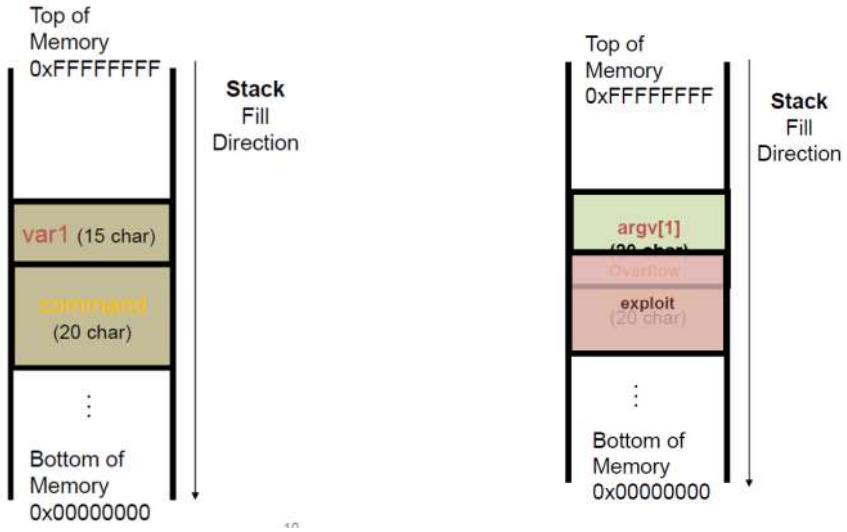


Figura 6.8: Esempio Stack-Based buffer overflow

6.5.4 Stack Smashing Attack

In questo tipo di exploit l'attaccante sfrutta una vulnerabilità buffer overflow per sovrascrivere l'indirizzo di ritorno della routine corrente. Quando la routine termina il programma esegue il codice malevolo iniettato dall'attaccante, anziché riprendere il normale flusso di esecuzione. La difficoltà di quest'attacco consiste nel fatto che l'attaccante deve conoscere l'esatta posizione sullo stack dell'indirizzo di ritorno. Al fine di confezionare un attacco efficace l'attaccante deve, infatti:

- assicurarsi che il codice malevolo iniettato risieda nell'address space del processo attaccato (altrimenti non sarebbe eseguito). Il codice può essere tenuto nel buffer stesso (payload)
- conoscere l'indirizzo esatto del codice malevolo
- e riuscire a sovrascrivere l'indirizzo di ritorno con l'indirizzo del codice malevolo Per svolegere quest'ultimo task sono disponibili diverse tecniche, che vedremo nei paragrafi seguenti.

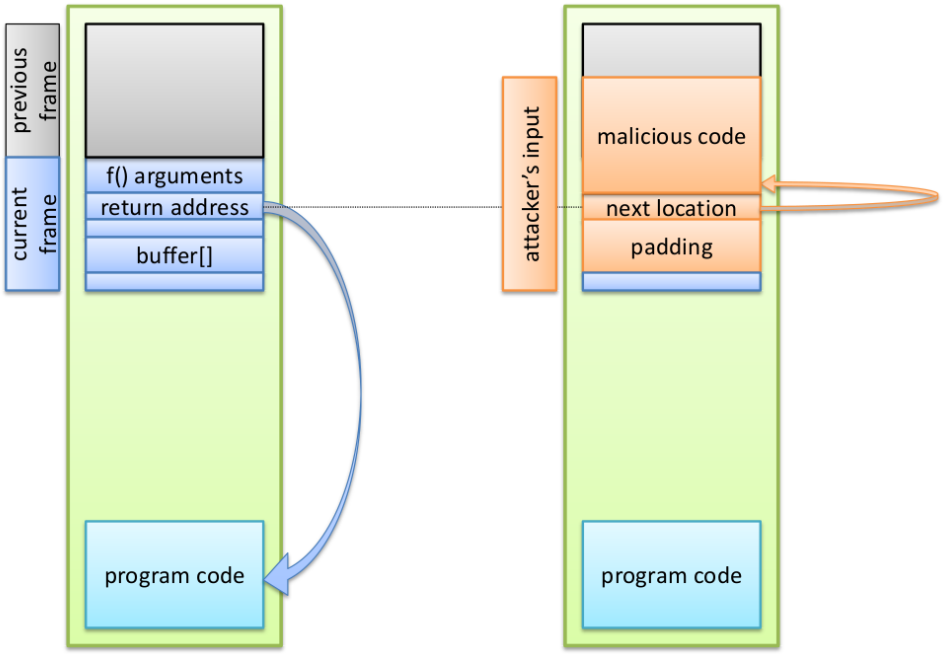


Figura 6.9: Esempio Stack-Smashing

6.5.5 NOP Sledding

Al fine di semplificare la stima dell'indirizzo del codice malevolo (da posizionare nel return address dello stack) si può incrementare la dimensione del payload inserendo un gran numero di operazione **NOP (No-OP)**. Un'istruzione di questo tipo non fa nulla e il processore passa all'istruzione seguente. Se l'attacco funziona il processo salterà al punto di ritorno stimato (sovrascritto da qualche operazione NOP) e "slitterà" attraverso le istruzioni NOP verso il codice malevolo. L'attaccante deve confezionare quindi un input che contiene:

- una quantità di dati appropriata da eccedere le dimensioni del buffer
- una stima dell'indirizzo di ritorno
- una grande quantità di istruzioni NOP

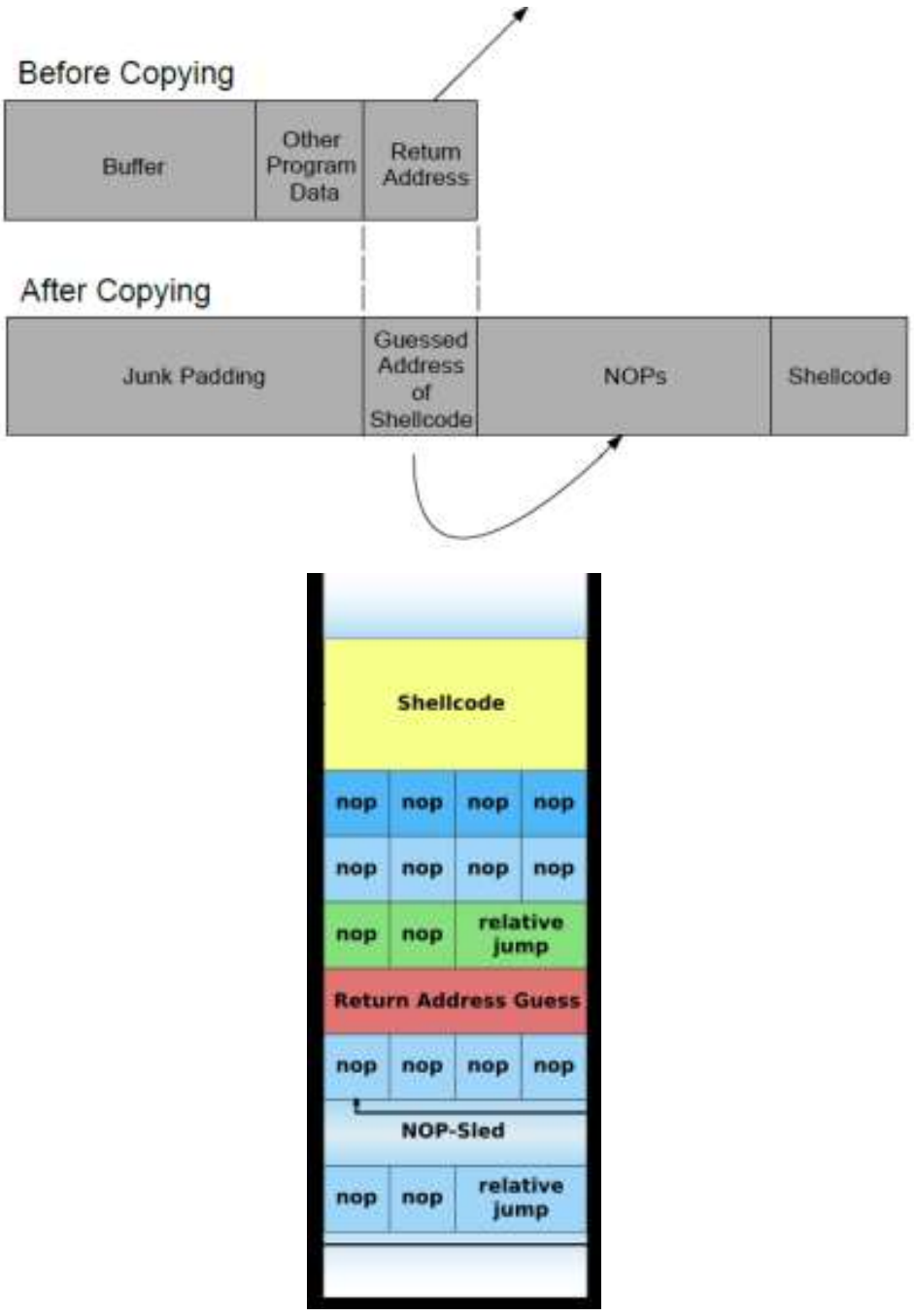


Figura 6.10: NOP Sledding

- lo shellcode

Il NOP-Sledding aumenta le probabilità di riuscita per un attacco buffer overflow. E' un attacco molto utilizzato, anche se presenta ancora diverse difficoltà: è difficile da automatizzare, serve molto spazio per il payload e bisogna ancora stimare, in qualche misura, l'indirizzo dello shellcode.

6.5.6 Trampolining

Questo tipo di attacco sfrutta la previdibilità dell'allocazione della memoria di librerie esterne. Al momento della loro inizializzazione molti processi caricano in zone protette del loro address space tali librerie. Un attaccante può sfruttare la conoscenza di una libreria di sistema per eseguire un attacco senza dover conoscere l'indirizzo dello shellcode. Ad esempio, se si è a conoscenza che una DLL di Windows richiede al processore di saltare all'indirizzo del registro ESP che punta ad un buffer, si può inserire del codice malevolo nel buffer indirizzato dal registro e sovrascrivere il punto di ritorno della funzione corrente con quello dell'istruzione nota della DLL. E' un attacco abbastanza complesso, ma molto pericoloso in quanto automaizzabile.

6.5.7 Return to libc

Anche questa tecnica sfrutta librerie esterne caricate in fase di esecuzione. In questo caso vengono utizzate le librerie C, chiamate **libc**. L'idea consiste nel determinare l'indirizzo di una funzione presente nelle librerie C all'interno dell'address space da colpire e forzare il programma a chiamare questa funzione. Nelle librerie C esistono diverse funzioni vulnerabili a questo tipo di attacco (e.g. exec(), system()). L'attacco consiste nel:

- sovrascrivere l'indirizzo di ritorno della funzione corrente con la funzione di libc da richiamare
- sovrascrivere il buffer con gli argomenti per la funzione chiamata (e.g. con "bin/sh")

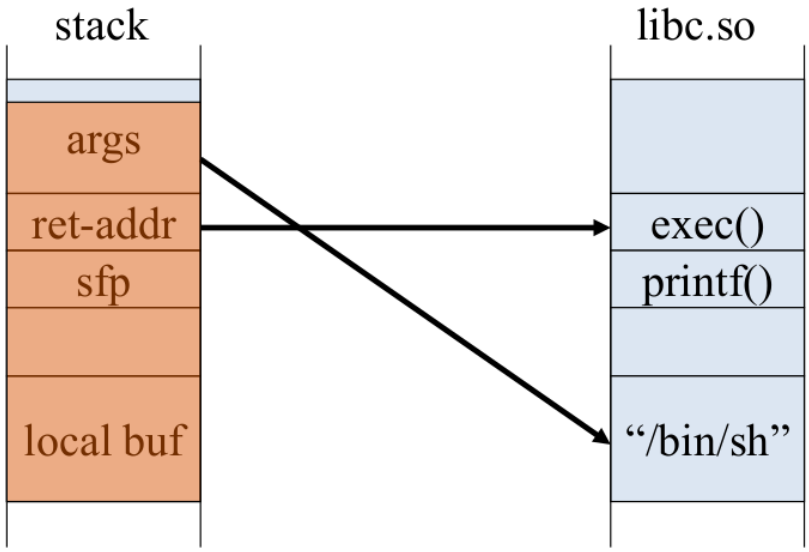


Figura 6.11: Esempio return to libc

Il principale vantaggio di questa tecnica consiste nel fatto che non viene eseguito alcun codice sullo stack, rendendo inefficaci le contromisure che rendono lo stack non eseguibili.

6.5.8 Contromisure Stack-Based BO

- Scrivere codice sicuro che verifica sempre le dimensioni dell'input proveniente dall'utente
- Utilizzare linguaggi di programmazione che non consentono questo tipo di attacchi. C e $C++$ sono vulnerabili, mentre $Java$, ad esempio, no, poiché gli oggetti vengono allocati dinamicamente sullo heap.
- Utilizzare meccanismi di protezione a livello di OS, come vedremo nei seguenti paragrafi

No eXecute bit - NX bit

Una possibile contromisura a questo tipo di attacchi è l'introduzione di un bit che marca i segmenti di memoria relativi allo stack e all'heap. In questo modo non è possibile eseguire shellcode direttamente in queste porzioni di memoria. Come accennato nel paragrafo precedente, questa contromisura **non** è efficace contro attacchi del tipo **return-to-libc**.

Address Space Layout Randomization - ASLR

Tecnica che consiste nell'arrangiare randomicamente l'address space di un processo. Ad esempio, due boot di Windows Vista danno luogo a locazioni delle libtrerie in memoria diverse, come mostrato in Figura 6.12.

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows
ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Figura 6.12: Effetto ASLR sull'indirizzo di librerie Windows

Stack-Smashing protection

Tecnica che prevede un controllo in fase di esecuzione. Nel momento in cui una routine chiama l'indirizzo di ritorno l'OS verifica che lo stack non sia stato modificato rispetto al momento in cui la funzione è stata chiamata. Se lo stack è stato modificato viene lanciato un errore di tipo **segmentation fault** e il programma termina forzatamente.

Canary

Anche questo metodo implementa un controllo in fase di esecuzione. Si definisce **Canary** un valore di controllo (spesso randomico) inserito dall'OS dopo un buffer o prima dell'indirizzo di ritorno. L'OS verifica regolarmente l'integrità di questo valore: in caso di buffer overflow viene sovrascritto anche il Canary, allertando l'OS.

6.5.9 Race Conditions

Una race condition è una situazione in cui il comportamento del programma è (involontariamente) dipendente dalla tempistica in cui si verificano certi eventi. Un classico esempio di queste situazioni fa uso delle seguenti due funzioni C:

- la funzione open() apre il file specificato utilizzando l'userID effettivo (piuttosto che l'userID reale) del processo chiamante per verificarne i permessi. In altre parole, se un programma setuid di proprietà dell'utente root è lanciato da un utente normale, il programma può

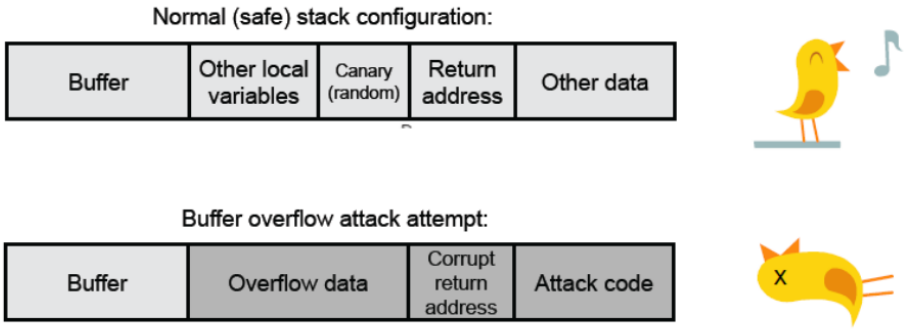


Figura 6.13: Canary

chiamare con successo open() sui file che solo l'utente root ha il permesso di accedere (SCRITTO DA CANI, RISCRIVERE)

- la funzione access() controlla se l'utente reale (in questo caso l'utente che esegue il programma) ha permesso di accedere al file specificato

Analizziamo ora l'esempio in questione. Supponiamo che un semplice programma richieda il nome di un file come argomento, controlli se l'utente che esegue il programma ha il permesso di aprire il file e in caso affermativo legga i primi caratteri del file e li stampi. C'è una race condition in questa implementazione: vi è un piccolo delay tra le chiamate ad access() ed a open(). Un utente malintenzionato potrebbe sfruttare questo piccolo ritardo, modificando il file in questione tra le due chiamate.

Ad esempio, supponiamo che l'attaccante richieda /home/joe/dummy come argomento, un file di testo innocente che il egli può accedere. Dopo che la chiamata access() restituisce 0, indicando che l'utente dispone dell'autorizzazione per accedere al file, l'utente malintenzionato può sostituire rapidamente /home/joe/dummy con un link simbolico a un file di cui non ha l'autorizzazione in lettura, come /etc/passwd. Successivamente, il programma chiamerà open() sul link simbolico, che avrà successo perché il programma setuid ha come proprietario root.

Si noti che questo tipo di attacco non potrebbe essere fatto manualmente: la differenza di tempo tra due chiamate di funzione è abbastanza piccola da fare in modo che nessun essere umano sia in grado di modificare il file in tempo. E' invece possibile avere un programma in esecuzione in background che scambia più volte i due file, ed esegue il programma vulnerabile finché lo scambio non si verifica esattamente tra le due istruzioni open() e access().

Time of Check/Time of Use Problem

In generale, questo tipo di vulnerabilità è conosciuto come Time of Check/Time of Use (**TOCTOU**) problem. Ogni volta che un programma controlla la validità e la le autorizzazioni per un oggetto, sia esso un file o di qualche altra proprietà, prima di eseguire un'azione su tale oggetto, occorre fare attenzione che queste due operazioni siano eseguite atomicamente (dovrebbero essere eseguite come una operazione unica). In caso contrario, l'oggetto può essere modificato tra il momento in cui viene controllato e quello in cui viene utilizzato.

Per rendere sicuro il codice dell'esempio la chiamata di access() dovrebbe essere completamente evitata. Il programma dovrebbe invece ritirare i propri privilegi usando setuid() prima

Listing 6.4: Esempio codice vulnerabile a race conditions

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char * argv[ ])
{
        int file;
        char buf[1024];
        memset(buf, 0, 1024);
        if(argc < 2) {
                printf("Usage: printer [filename]\n");
                exit(-1);
        }

        if(access(argv[1], R_OK) != 0) {
                printf("Cannot access file.\n");
                exit(-1);
        }

        file = open(argv[1], O_RDONLY);
        read(file, buf, 1023);
        close(file);
        printf("%s\n", buf);
        return 0;
}
```

di chiamare open(). In questo modo, se l'utente che esegue il programma non ha il permesso di aprire il file specificato, la chiamata open() fallirà.

Listing 6.5: Esempio codice sicuro

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char * argv[ ])
{
        int file;
        char buf[1024];
        uid t uid, euid;
        memset(buf, 0, 1024);
        if(argc < 2) {
                printf("Usage: printer [filename]\n");
                exit(-1);
        }

        euid = geteuid();
        uid = getuid();

        /* Drop privileges */
        seteuid(uid);
        file = open(argv[1], O RDONLY);
        read(file, buf, 1023);
        close(file);

        /* Restore privileges */
        seteuid(euid);
        printf("%s\n", buf);
        return 0;
}
```