UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria del Software II

# *Security testing semi-automatico di applicazioni basate su container*

Anno Accademico 2019/2020

Relatori
**Ch.mo prof. Porfirio Tramontana**
**Ch.ma prof.ssa Alessandra De Benedictis**
**Ch.ma prof.ssa Valentina Casola**

Candidato
**Giuseppe D'Alterio**
**matr. M63000880**

*A chi,*

*distinguendo poco fra lavorare e giocare,*

*ha saputo mettermi nelle condizioni migliori per imparare...*

# Introduzione

Le recenti metodologie di sviluppo software agile diventano ogni giorno sempre più popolari, specialmente per quanto riguarda lo sviluppo di servizi in esecuzione in container. Queste metodologie, infatti, riducono drasticamente il time-to-market ma, allo stesso tempo, risultano difficili da integrare con la progettazione della sicurezza e con le metodologie di gestione del rischio. Un approccio proattivo, come il testing di sicurezza, è fondamentale per identificare le minacce e le vulnerabilità cui sono affette le applicazioni prima che la loro sicurezza sia violata. Tra le tante tecniche di sicurezza, per identificare e provare l'esistenza di strategie efficaci di attacco, quella di maggior spicco è il penetration testing. Purtroppo, il penetration testing è, al momento, una procedura poco automatica, molto human-driven, che richiede una profonda conoscenza dei possibili attacchi da testare e degli strumenti di attacco più noti per l'esecuzione dei test. Insomma, la procedura richiede tempo, competenze e di conseguenza denaro. Per questo motivo, il più delle volte è eseguita solo al termine del processo di sviluppo quando oramai una modifica anche banale al software potrebbe

impattare molto e risultare assai costosa. Inoltre, il fatto che i moderni processi di sviluppo software agile spingano su di uno sviluppo a rilasci incrementali, a seguito di "sprint" di sviluppo e testing, accentua ulteriormente il problema. Queste considerazioni hanno ispirato la definizione di una metodologia di testing di sicurezza automatico, integrabile con i moderni processi di sviluppo software, volta a rendere semplice l'esecuzione di test di sicurezza a partire dai primi prototipi. Così facendo, anche uno sviluppatore con una limitata conoscenza in termini di sicurezza può ottenere una stima, seppur grossolana, del livello di sicurezza dell'applicazione in via di sviluppo.

La sicurezza del System under Test (SuT) viene valutata tramite un apposito processo automatico che va ad effettuare la configurazione e l'esecuzione di test di sicurezza a partire dalla descrizione (mediamente diagramma MACM) dell'applicazione e delle risorse utilizzate. Il processo, poi, costruisce un modello della sicurezza della applicazione e lo sfrutta come base di conoscenza per identificare e determinare i test di sicurezza da eseguire.

Quando si parla di processi agili e di DevOps si fa riferimento a metodologie che si fondano, seppur non direttamente, su una serie di tecnologie abilitanti tra le quali spicca Docker o, più in generale, i container. I container emergono come un'alternativa estremamente leggera alle macchine virtuali, offrendo un miglior supporto alle architetture a microservizi. Sebbene siano oramai considerati il metodo standard

per la distribuzione di servizi software (specialmente microservizi), le indagini di mercato mostrano come, in molte aziende, la sicurezza dei container sia la principale preoccupazione per la loro adozione [30] .

L'obiettivo di questo lavoro è quello di proporre un'estensione alla suddetta metodologia di testing di sicurezza, specializzata allo sviluppo di applicazioni in esecuzione all'interno di container, oltre che presentare uno strumento automatico che implementa quanto proposto.

Infine, viene presentato un utilizzo dello strumento con delle applicazioni di esempio, in esecuzione in un ambiente a container. L'obiettivo è quello di mostrare quanto lo strumento sia semplice da utilizzare, anche da parte di personale inesperto nel campo della security, e, più in generale, di aiuto per la gestione e la valutazione della sicurezza di un sistema software.

# Indice

**Introduzione** **ii**

**1 La sicurezza nel processi di sviluppo software** **1**

1.1 La sicurezza nelle metodologie di sviluppo software . . 1

1.2 Metodologie di Security-By-Design . . . . . 2

1.3 La sicurezza nelle metodologie agili . . . . . 4

**2 Security testing semi-automatico basato su Security-by-Design** **6**

2.1 Formalismo MACM per la descrizione del sistema . . . 7

2.2 Modello di sicurezza . . . . . 8

2.3 Il processo di penetration testing . . . . . 11

2.4 La fase di Preparazione . . . . . 12

2.4.1 Configurazione ambiente di testing e analisi del rischio . . . . . 13

2.4.2 Arricchimento del modello di sicurezza . . . . . 13

2.5 La fase di Scansione . . . . . 14

2.6 La fase di Penetration testing . . . . . 14

**3 La sicurezza dei container** **15**

3.1 Modello dei threat . . . . . 16
3.2 Sicurezza in Docker . . . . . 20
3.2.1 I container Docker . . . . . 21
3.2.2 Analisi della sicurezza in Docker . . . . . 22
3.3 Estensione al formalismo MACM per container . . . . 31

**4 Realizzazione di uno strumento di testing automatico 35**
4.1 Workflow implementato . . . . . 36
4.1.1 Scansione del sistema . . . . . 36
4.1.2 Generazione del modello di sicurezza . . . . . 37
4.1.3 Testing di sicurezza . . . . . 39
4.1.4 Generazione report . . . . . 39
4.2 Architettura dello strumento . . . . . 43
4.2.1 File prodotti . . . . . 44
4.3 Sviluppo . . . . . 46
4.3.1 Struttura componenti e relazioni . . . . . 46
4.3.2 Estendere lo strumento . . . . . 48
4.4 Funzionalità aggiuntive . . . . . 52
4.4.1 Evoluzione di una rilevazione . . . . . 52
4.4.2 Evoluzione dello stato complessivo . . . . . 53
4.4.3 Statistiche . . . . . 55
4.5 Utilizzo dello strumento . . . . . 56
4.5.1 Utilizzo avanzato dello strumento . . . . . 57
4.6 GitHub Action . . . . . 57

**5 Caso di studio: Juice Shop 61**
5.1 Configurazione ambiente . . . . . 62

5.2 Descrizione del sistema mediante il formalismo MACM 63
5.3 Configurazione dello strumento . . . . . 64
5.4 Analisi cicli di evoluzione del sistema . . . . . 64
5.4.1 Applicazione delle mitigazioni proposte v.11.1.3-1 67
5.4.2 Applicazione ulteriori modifiche al sistema v.11.1.3-2 . . . . . 71
5.4.3 Applicazione modifiche al sistema, con regressione v.11.1.3-3 . . . . . 72
5.5 Analisi andamento storico generale . . . . . 74
5.6 Analisi andamento storico rilevazioni . . . . . 75
5.7 Discussione risultati . . . . . 76

**6 Caso di studio: Sito web basato su Joomla! 78**
6.1 Configurazione ambiente . . . . . 80
6.2 Descrizione del sistema mediante il formalismo MACM 80
6.3 Configurazione dello strumento . . . . . 81
6.4 Analisi cicli di evoluzione del sistema . . . . . 82
6.4.1 Analisi nuova versione del sistema . . . . . 85
6.5 Discussione risultati . . . . . 87

**7 Conclusioni 88**

**8 Sviluppi futuri 91**

# Capitolo 1

# La sicurezza nel processi di sviluppo software

## 1.1 La sicurezza nelle metodologie di sviluppo software

L'analisi degli aspetti di sicurezza nella progettazione dei sistemi software è al centro delle pratiche di ingegneria della sicurezza, mirate alla costruzione di sistemi robusti a possibili interruzioni, minacce e pericoli. Queste pratiche, tipicamente, suggeriscono l'adozione di processi da applicare in maniera sistematica al sistema target e da eseguire durante l'intero ciclo di vita, soffermandosi maggiormente sulle attività di testing post-sviluppo, mirate a convalidare l'efficacia dei controlli di sicurezza già applicati o ad identificare gli attuali punti di debolezza

(weakness) del sistema, così da guidare gli investimenti nei confronti della sicurezza.

Un approccio ampiamente seguito in letteratura, dell'ingegneria della sicurezza, è quello di dedicarsi alla sicurezza, durante il ciclo di vita del software, tramite i principi dell'ingegneria dei requisiti. Più nel dettaglio, in questo contesto è stato presentato un framework [4] per modellare ed analizzare i requisiti di sicurezza, separando gli attori che manipolano le risorse da quelli che le possiedono, così da verificare i requisiti tramite un'apposita logica di delega. Sono presenti in letteratura anche ulteriori proposte di approcci simili ma comunque ugualmente onerosi in termini di tempo e denaro, poiché richiedono una profonda competenza nel campo della sicurezza e si basano su procedure formali e complesse che impattano negativamente sulle timeline dei processi di sviluppo software. Per queste ragioni, i suddetti processi sono applicati nei contesti security-critical ma spesso trascurati nelle ottiche di sviluppo software di medio-piccole imprese poiché ritenuti a malapena dei requisiti aggiuntivi per i loro prodotti.

## 1.2 Metodologie di Security-By-Design

Per Security-By-Design [9] si intende un approccio alternativo ai principi di Security-by-Obscurity, fondati sul mantenere segreta e confidenziale l'implementazione del sistema, al fine di mantenere il sistema stesso in sicurezza. L'approccio di Security-By-Design suggerisce, in-

vece, l'adozione di misure proattive nei confronti delle minacce note e l'implementazione di paradigmi di Secure-by-Default [10] nella configurazione dei componenti software e delle politiche di accesso. La Security-By-Design richiede interesse agli aspetti di sicurezza fin dalla progettazione del sistema, anche tramite, ad esempio, l'adozione di hardware fidato. I processi inoltre includono, un'approfondita threat analysis, la progettazione di contromisure per le minacce note esistenti, come parte integrante dell'architettura, e l'esecuzione di rigorosi test di sicurezza.

Sulla base di questo principio, sono stati definiti numerosi processi di sviluppo sicuro (*SDL*). I più comuni sono Microsoft SDL, OWAS OpenSAMM e Cisco SDL. Tutti questi includono la modellazione dei threat come step iniziale del processo e continui test di sicurezza, oltre che l'assessment, durante le varie fasi di sviluppo software. Tra i principali limiti di questo approccio ritroviamo, ancora una volta, i costi: c'è la necessità di avere un team di esperti di sicurezza, sia in termini di progettazione che di sviluppo che di test. Difatti, proprio per questo motivo, pochissime aziende attualmente applicano questa metodologica per lo sviluppo delle loro applicazioni, preferendola solamente nel caso di applicazioni security-critical.

## 1.3 La sicurezza nelle metodologie agili

Sebbene siano stati definiti diversi approcci, come ad esempio quelli presentati in precedenza, per lo sviluppo di applicazioni sicure, nessuno di questi riesce ad integrarsi bene nei moderni processi di sviluppo come DevOps o Agile. Questi ultimi, infatti, basano il loro successo su rilasci incrementali del software, rapidi ed automatici, a seguito di un processo di sviluppo e test molto rapido.

In letteratura è comunque possibile ritrovare proposte per l'integrazione della gestione della sicurezza nelle metodologie agili. Azham et al. [40], ad esempio, hanno proposto un'estensione alla metodologia Scrum che si avvale di "backlogs di sicurezza" da utilizzare nelle varie fasi di Scrum. Ad ogni modo, non sono previste fasi specifiche per la definizione dei requisiti di sicurezza e per la risk assessment e, inoltre, i vincoli di sicurezza dettati sono facilmente rotti dal processo di sviluppo incrementale in cui il codice cambia frequentemente [38] [15] [24] [26].

Una ulteriore proposta è stata quella di "S-Scrum" [21] che incorpora attività di analisi e modellazione di sicurezza in cosiddetti "spikes", effettuati dopo il realese planning. Gli autori, ad ogni modo, non propongono alcuna tecnica di analisi bensì suggeriscono l'adozione di tecniche di penetration testing negli stage di test.

Infine, Pohl et al. [27] hanno proposto "Secure Scrum", un framework che non va ad alterare il tradizionale processo Scrum ma va a

migliorare le user stories con l'aggiunta di tags relative alla sicurezza. Secure Scrum è progettato per assicurare che venga raggiungo un adeguato livello di sicurezza, ed assume che la maggior parte dei requisiti siano gestiti dal team stesso, ammettendo tuttavia la partecipazione di consulenti di sicurezza esterni per particolari problemi.

Recenti studi evidenziano come gli strumenti di automazione DevOps, se correttamente utilizzati, possano essere utilizzati per affrontare correttamente procedure di security assessment [20] [36], dando vita a processi security-oriented come SecDevOps e DevSecOps. Tuttavia, allo stato attuale, non vi sono ancora delle metodologie adatte ad affrontare un processo automatico di assessment di sicurezza.

Le moderni metodologie di sviluppo esortano metodi e tecniche in grado di integrare la sicurezza in ogni iterazione di sviluppo, preferibilmente basati su approcci di Security-by-Design, e che non impattino negativamente sulla flessibilità e velocità di sviluppo. Inoltre i task devono assolutamente essere piccoli, eseguibili da personale con competente basiche di sicurezza e, perché no, automatici.

Per questo motivo, il presente lavoro mira a proporre tecniche e strumenti automatici utilizzabili anche da personale non esperto.

# Capitolo 2

# Security testing semi-automatico basato su Security-by-Design

Tra le attuali soluzioni al testing di sicurezza di un applicazione vi è l'uso di tecniche di penetration testing. Queste tecniche sono solitamente manuali e richiedono una profonda conoscenza dei possibili attacchi attuabili e degli strumenti di testing in grado di lanciare attacchi noti in poco tempo, La strategia di testing di sicurezza proposta in seguito, facilmente integrabile con le pratiche di CI/CD, mira ad essere automatica e di semplice utilizzo [6] .

Il processo si basa sulla descrizione del System under Test tramite formalismo MACM (Paragrafo 3.3), per poi proseguire con attività di

scansione e testing, guidato dal modello presentato al Paragrafo 2.2, per fornire le informazioni necessarie ad orchestrare gli strumenti di penetration testing.

## 2.1 Formalismo MACM per la descrizione del sistema

Il formalismo MACM (Multi-cloud Application Composition Model) [7], descrive i componenti logici che costituiscono il sistema e le loro relazioni mediante un grafo.

Le tipologie di nodi del sistema sono:

- **Software as a Service**, modella il software in esecuzione
- **Infrastructure as a Service**, modella una virtual machine
- **CSP**, modella il provider che offre la IaaS

Per quanto riguarda gli archi:

- Un nodo CSP può fornire (*provides*) uno o più VM
- Un nodo IaaS può hostare (*hosts*) uno o più SaaS
- Un nodo SaaS può utilizzare (*uses*) altri SaaS.

In aggiunta alle informazioni in merito all'architettura dell'applicazione, il formalismo MACM consente di specificare anche informazioni di

dettaglio per l'accesso al componente, come ad esempio l'indirizzo IP ed il porto per accedere al servizio.

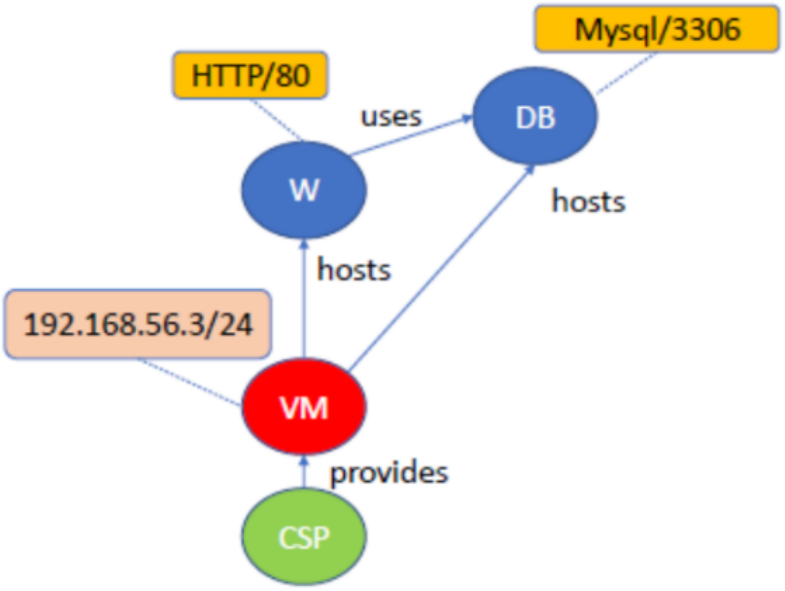


Figura 2.1: Esempio modellazione MACM

## 2.2 Modello di sicurezza

Il modello di sicurezza presentato è mirato alla generazione di una valutazione a grana grossa delle vulnerabilità cui un'applicazione, ad esempio in esecuzione in un container, è esposta attraverso l'esecuzione automatica di test suite di penetration testing ben specifiche.

Il modello fa uso dei principali concetti di sicurezza come *threats*, *exploits*, *attacks*, *weaknesess* e *vulnerabilities* cui si intende:

- *Threat*: un'attività, intenzionale o meno, con il potenziale di poter causare danni ad un sistema di gestione delle informazioni

- *Threat agent*: l'entità che da vita ad un threat e che esegue un attacco verso un asset
- *Exploit*: sfruttare una vulnerabilità di un sistema per generare comportamenti non previsti
- *Attack*: tentativo di visualizzare, alterare, disattivare,, distruggere rubare o fornire accesso/utilizzo non autorizzato ad un asset
- *Weakness*: una tipologia di errore in un sistema che, sotto certe condizioni, potrebbe contribuire all'introduzione di una vulnerabilità
- *NIST Security Control*: controllo di sicurezza, dettato dal NIST 800-52, che viene a mancare se la weakness associata è presente nel sistema
- *Vulnerability*: un difetto in un sistema che può essere sfruttato per violarne la sicurezza

Il modello è popolato con i dati ottenuti da diverse sorgenti, standard e non, tra cui:

- OWASP
- Mitre CWE
- Mitre CVE
- Mitre CAPEC

- NIST 800-53

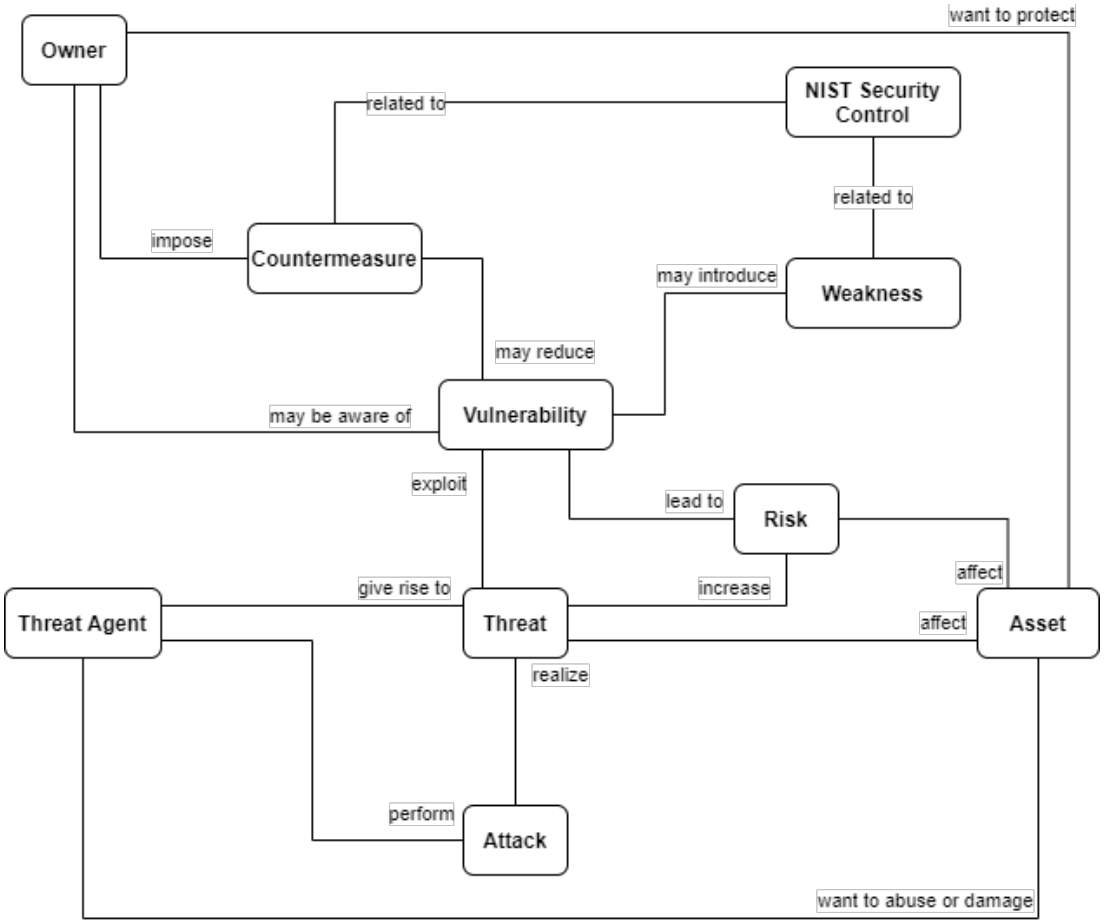


Figura 2.2: Modello della sicurezza

Il modello mostrato in Figura 2.2 illustra gli aspetti principalmente coinvolti nella caratterizzazione della sicurezza di un sistema, collegando tra loro diversi concetti introdotti dagli standard di sicurezza. Tra queste troviamo la "Common Criteria for Information Technology Security Evaluation" (ISO 15408), la ISO 27000 ovvero la famiglia di norme SGSI, "Sistemi di Gestione per la Sicurezza delle Informazioni", che raggruppa un insieme di norme internazionali che si prefiggono di proteggere le informazioni che vengono mantenute ed elaborate da

un'organizzazione, e la "Common Weakness Enumeration" (CWE) di Mitre, cui è appunto tratta la definizione di *weakness*.

Con l'obiettivo di agevolare e semplificare un eventuale processo di assessment di sicurezza del sistema, viene esteso il modello originale con l'aggiunta del controllo di sicurezza NIST 800-53 [22] associato alla specifica weakness, oltre che alla contromisura. Il mapping è fornito da MITRE [19].

## 2.3 Il processo di penetration testing

La strategia proposta di penetration testing implica l'esecuzione dei test mirati alla verifica della possibilità di exploitare un threat o una vulnerabilità, tra quelle presenti nel sistema. Il processo di testing semi-automatico consiste di 3 fasi, illustrate in Figura 2.3:

1. **Preparazione**, configurazione ambiente di test ed identificazione del System under Test

2. **Scansione**, identificazione delle weakness e delle vulnerabilità

3. **PenTesting**, configurazione ed esecuzione degli attacchi

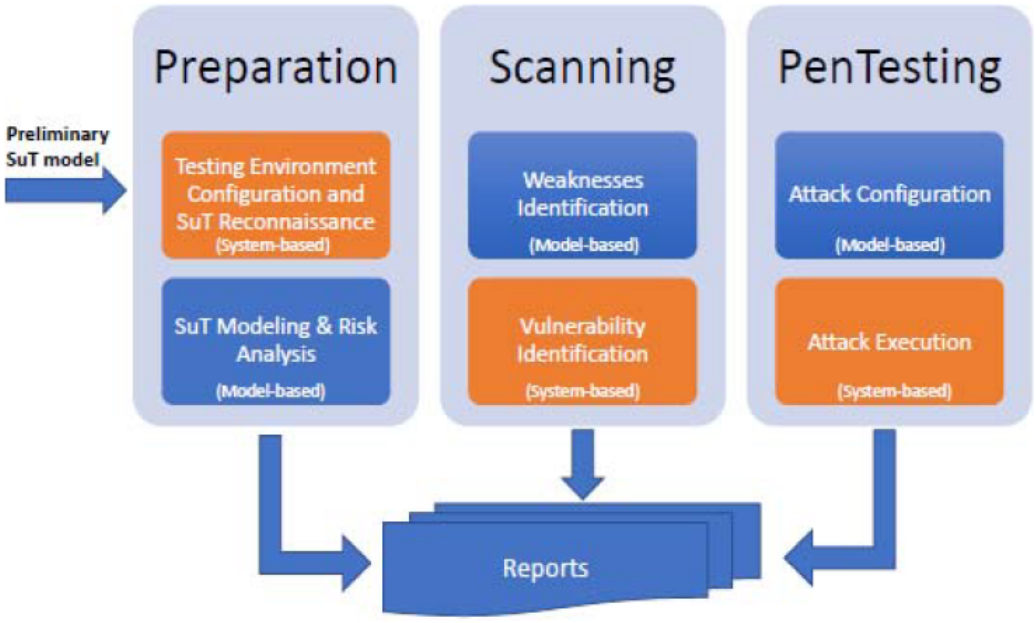


Figura 2.3: Processo di penetration testing semi-automatico

## 2.4 La fase di Preparazione

La fase di Preparazione prende in input una descrizione preliminare del System under Test, basata sulla conoscenza iniziale del sistema. In questa fase, le attività model-based hanno il compito di determinare i threats ed analizzare i rischi annessi da ciascun threat. Le attività system-based, invece, si occupano della configurazione dell'ambiente di testing.

### 2.4.1 Configurazione ambiente di testing e analisi del rischio

L'automazione del processo di penetration testing è infatti possibile grazie alla presenza di un ambiente di testing pre-configurato, automaticamente messo a punto per le specifiche esigente delle attività system-based.

### 2.4.2 Arricchimento del modello di sicurezza

Il modello preliminare del System under Test viene arricchito con le informazioni raccolte dai vari strumenti utilizzati per la scansione del SuT. Resta comunque da sottolineare il fatto che, sebbene gli strumenti automatici di scan siano in grado di rilevare i componenti software, l'intervento umano resta necessario per mappare questi componenti ai servizi. Sulla base del modello arricchito, viene eseguito un processo di analisi del rischio, con l'approccio presentato in Casola et al. (2016) [8]. Questo processo consente l'identificazione dei principali threat cui ogni componente è soggetto, basandosi sulle informazioni specifiche dell'implementazione e sull'interazione con le altre componenti. Successivamente il tester è guidato in un processo di classificazione del rischio basato sulla "OWASP Risk Rating methodology" (The OWAS Foundation, 2018) che consente di identificare e classificare i rischi esistenti, valutati come $Risk = ProbabilityOfOccurrence \cdot Impact$, raggruppandoli nelle categorie definite dalla "STRIDE categorisation

methodology" (Microsoft Corporation, 2016): *spoofing*, *tampering*, *repudation*, *information disclosure*, *denial of service* and *elevation of privileges*.

## 2.5 La fase di Scansione

La fase di Scansione è dedicata all'identificazione delle weakness e delle vulnerabilità che interessano il System under Test. Le weakness sono identifiacate come parte delle attività model-based, mentre le vulnerabilità sono invece estrapolate da un processo di scan attivo. Questa fase di scan avviene con l'ausilio di particolari strumenti di scan come ad esempio OpenVAS, OWASP ZAP, Anchore Container Analysis etc.

## 2.6 La fase di Penetration testing

La fase di penetration testing consiste nella progettazione ed esecuzione automatica di attacchi mirati all'exploiting delle weakness e vulnerabilità identificate nella fase precedente. Nello specifico, gli attacchi sono preparati tramite una attività model-based per identificare ed orchestrare gli attacchi, quest'ultimi eseguiti lanciando appositi strumenti di penetration testing (*Metasploit*, *ZAP*, etc.).

# Capitolo 3

# La sicurezza dei container

Le macchine virtuali offrono una buon sicurezza, tuttavia l'overhead prestazionale introdotto, specialmente nelle macchine server cui si è soliti ritrovare numerose macchine virtuali in esecuzione è decisamente elevato. Difatti, ogni macchina virtuale mantiene la sua copia del sistema operativo, risorse dedicate, librerie ed applicazioni. Questo ha effetti dannosi sulle performance e sulle dimensioni di storage.

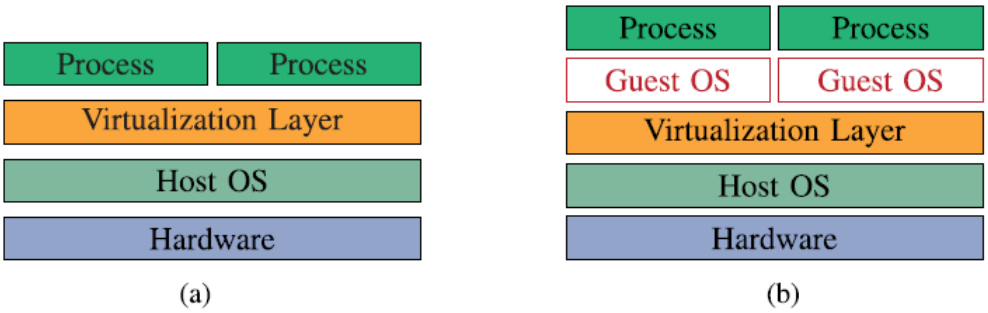


Figura 3.1: Confronto tra container e hypervisor - (a) Container, (b) Macchina virtuale

Con l'avvento dei processi di sviluppo DevOps ed il sempre più frequen-

te utilizzo delle architetture a microservizi, le VM rappresenterebbero un bottleneck prestazionale nel caso di deploy di ciascun microservizio in una VM separata. E' qui che ritroviamo il motivo del successo della virtualizzazione container-based: un'alternativa molto leggera alle macchine virtuali. I container possono condividere lo stesso kernel, riducendo tmepi di avvio e risorse richieste da ciascuna immagine. Recenti studi mostrano che un container può richiedere, per avviarsi, anche solo lo 0,12% del tempo di avvio richiesto da una tradizionale macchina virtuale [17]. Sebbene grazie alla loro leggerezza i container siano oramai considerati lo standard per il deploy di microservizi e applicazioni cloud [31], i container sono meno sicuri delle macchine virtuali [37] [33].

## 3.1 Modello dei threat

Nella trattazione si considera un host $H$ che ha $n = |C|$ containers ($C = c_1, c_2, ..., c_n$), ciascuno dei quali esegue una ed una sola applicazione, ipotizzando che tutti gli host abbiano risorse limitate e siano quindi vulnerabili ad attacchi mirati alla loro *availability*. Il modello dei threat [30] è dettagliato in Tabella 3.2.

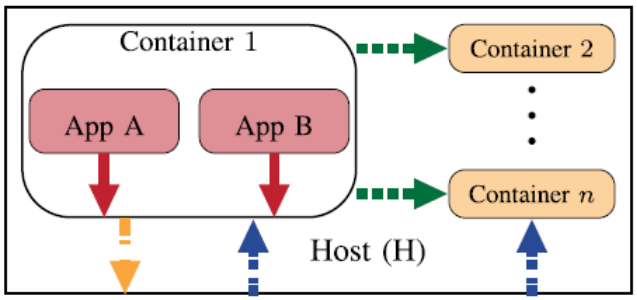


Figura 3.2: Potenziali attacchi intra/inter container e host

In Figura 3.2 ritroviamo le tipologia di attacchi intracontainer, intercontainer e tra container e host:

- Freccia rossa, l'applicazione attacca il container in cui vive
- Freccia verde, un container attacca un altro container
- Freccia gialla, un container attacca l'host
- Freccia blu, l'host attacca un container

Facendo l'ipotesi che ogni container abbia solo un'applicazione in esecuzione e che le applicazioni utente siano "oneste", i casi analizzati saranno di "un container che attacca un altro container" e "host che attacca un container". Sulla base del modello in Tabella 3.2, verrà opportunamente popolato il modello di sicurezza proposto al Paragrafo 2.2.

| **Threat** | **Attacco** | **Scenario** | **Soluzione** |
|---|---|---|---|
| Applicazione malintenzionata | Attacco DoS verso altri container | Un'applicazione malintenzionata può sferrare attacchi DoS verso altri container (e.g. syn floog, ICMP flood) oppure utilizzare utilizzare tante risorse dell'host per impattare sulle performance degli altri container | Scansione periodica delle vulnerabilità per immagini e applicazioni |
| Errata configurazione container | Esecuzione codice remoto, DoS | CVE-2014-9357, CVE-2014-6407 sono esempi di vulnerabilità in Docker per l'esecuzione di codice remoto e per attacchi DoS | I container in esecuzione dovrebbero essere monitorati in cerca di vulnerabilità, ed aggiornati, periodicamente |

| **Threat** | **Attacco** | **Scenario** | **Soluzione** |
|---|---|---|---|
| Impropria configurazione della gestione dell'accesso alla rete per i containers | Scanning di porti e vulnerabilità | Se un container compromesso ha accesso alla rete di altri container, può utilizzarla per ottenere informazioni sulle vulnerabilità note e sui porti aperti | I containers dovrebbero condividere la rete solo necessario e, inoltre, l'accesso alla rete deve essere configurato secondo il principio del minimo privilegio. Inoltre è consigliato l'uso di strumenti per il monitoraggio delle anomalie |
| Traffico inter-container non separato | ARP spoofing (MITM), MAC flooding, DoS | Un traffico non separata tra i containers consente ai container di sferrare attacchi MITM, DoS e MAC flooding [3] | I containers non dovrebbero essere in grado di comunicare se non necessario. Configurazione della rete secondo il principio del privilegio minimo. Implementazione di algoritmi per la prevenzione di attacchi DoS in ambienti a Containers [16]. |

Tabella 3.2: Tabella Threat inter-container

## 3.2 Sicurezza in Docker

Docker è tra i principali progetti per la containerizzazione delle applicazioni. La piattaforma è stata rilasciata come progetto open source per automatizzare il deployment, l'installazione e la messa in esercizio di applicazioni e/o sistemi software, sotto-forma di container.

I container Docker possono essere eseguiti in qualsiasi ambiente, purché questo supporti Docker, in maniera totalmente indipendente dall'hardware [35] .

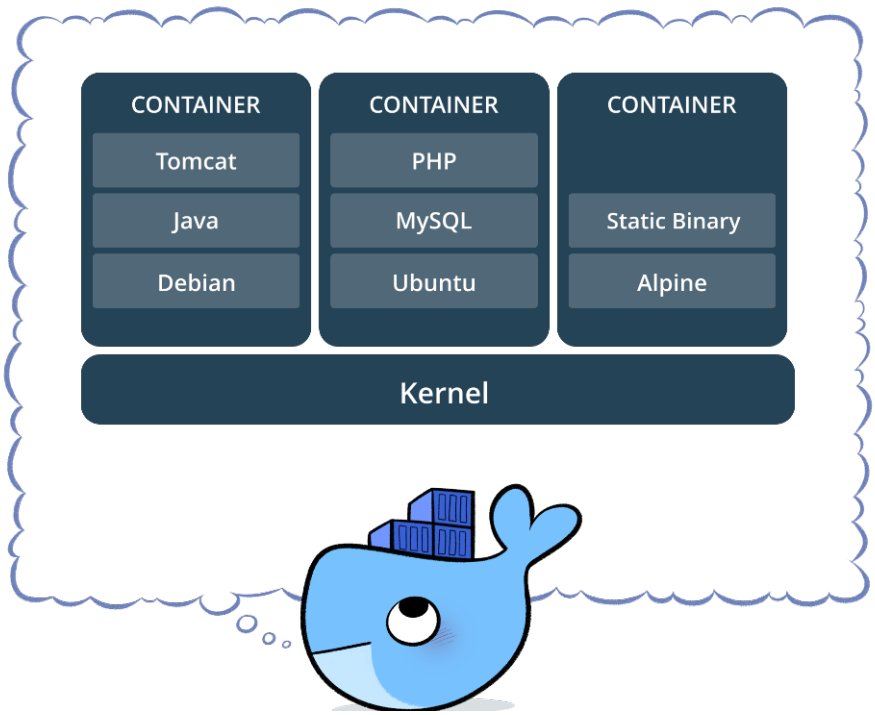


Figura 3.3: Docker

### 3.2.1 I container Docker

La prima versione di Docker (Marzo 2013) utilizzava LXC come ambiente di esecuzione predefinito, tuttavia col tempo LXC è stato abbandonato per fare spazio alla libreria *libcontainer*.

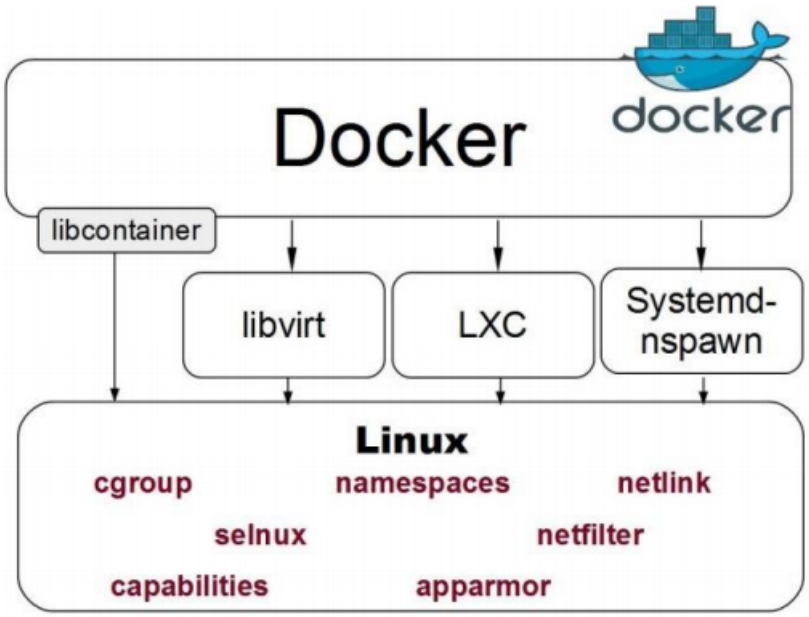


Figura 3.4: Architettura Docker

Le immagini Docker, ovvero un file strutturato in strati per eseguire codice (come ad esempio intere applicazioni) in container, sono condivisibili tramite dei *Docker registry* che possono essere locali, remoti, privati o pubblici. Tra i più importanti troviamo il *Docker Hub*, il repository pubblico ufficiale cui sono presenti le immagini dei SaaS più noti.

Per il rilascio di un'immagine di una propria applicazione è necessario seguire una procedura dedicata che prevede la build dell'immagine Docker, il push della stessa su di un repository Docker ed il pull per l'esecuzione su di un altra macchina, come illustrato in figura 3.5.

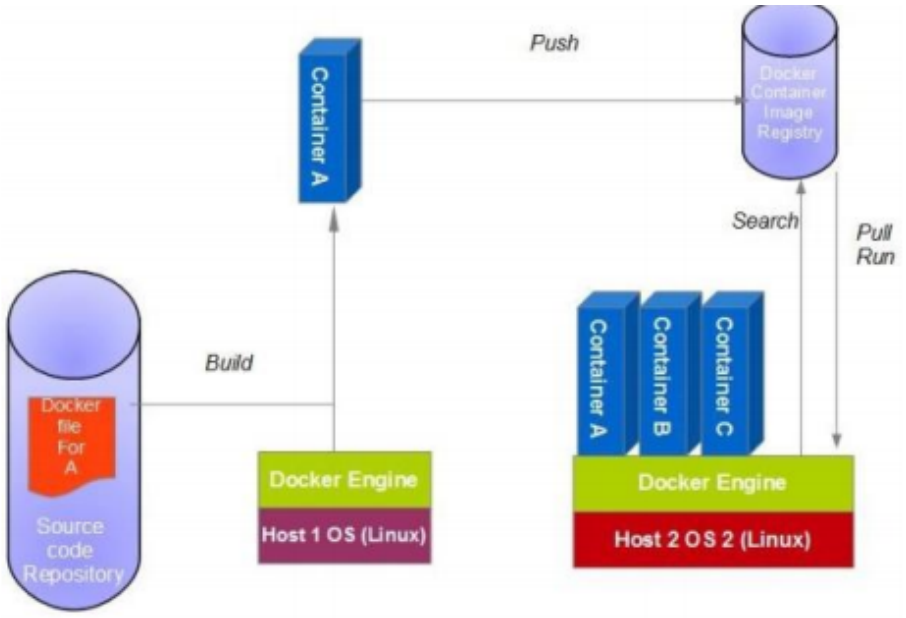


Figura 3.5: Sviluppo e rilascio di un container Docker

### 3.2.2 Analisi della sicurezza in Docker

Docker si presenta come l'alternativa leggera alla virtualizzazione classica, offrendo alle applicazioni containerizzate lo stesso kernel del sistema operativo della macchina host al fine di minimizzare l'overhead prestazionale, sebbene questo possa esporre i containers, se non op-

portunamente configurati, a numerosi rischi in termini di sicurezza [18].

Le applicazioni in esecuzione su tradizionali macchine virtuali (VM) possono comunicare solo con il kernel della propria VM e non con quello della macchina host. Al contrario, le applicazioni container-based possono comunicare direttamente con il kernel dell'host: questo è una delle principali debolezze per la sicurezza e la gestione delle privacy in un contesto a container [13] [12].

Per analizzare le vulnerabilità di Docker, suddividiamo gli attacchi in base a dove originano rispetto al sistema:

- Gli attacchi **outsider** provano, dall'esterno, ad ottenere l'accesso ad un container, cercando di danneggiare i dati o, comunque, di exploitarne le risorse
- Gli attacchi **insider** provano ad ottenere l'accesso diretto ai comandi di Docker, partendo dall'interno del sistema.

L'obiettivo di entrambi resta comunque quello di danneggiare i container in esecuzione e di rubare e/o manipolare dati sensibili.

Un esempio di semplice comando che, se eseuito, riesce a danneggiare gravemente un sistema non opportunamente configurato è il seguente:

$$dockerrun --privileged -v/usr : /usrbusyboxrm -rf/usr$$

In sostanza, il comando cancella la cartella */usr* dal file system della macchina host.

L'obiettivo di questa analisi è quello di trovare le maggiori vulnerabilità (sia insider che outsider) e suggerire delle precauzioni e strategie di mitigazione per contrastare potenziali attacchi.

Ad esempio, avviare il servizio Docker con il flag $-selinux$ abilità *SELinux*, un'architettura di sicurezza per i sistemi Linux, che offre agli amministratori un livello di controllo superiore sugli utenti autorizzati ad accedere al sistema [14].

Vediamo dunque più nel dettaglio differenti attacchi praticabili e relative proposte per mitigarli.

### Kernel Exploits

Il kernel della macchina host, gestisce le operazioni dei container e dei processi in esecuzione sia in Docker che sulla macchina host stessa. Nel caso di un exploit kernel-level, come detto in precedenza, le applicaizoni in esecuzione in containers possono interagire direttamente con il kernel della macchina host, condiviso da tutti i containers [5]. Pertanto, un attacco di questo genere va ad impattare tutti i containers in esecuzione e, potenzialmente, anche la macchian host. Per questa tipologia di exploits, Docker suggerisce di utilizzare *AppArmor* e *SELinux* quando Docker Engine è in esecuzione: così facendo si riesce ad avere un maggior controllo sulle specifiche autorizzazioni di un utente/pro-

cesso. Inoltre, non bisogna eseguire con i diritti di root i containers di cui non ci si fida pienamente.

Anche la comunicazione inter-container dovrebbe essere disattivata ed i containers di cui ci si fida, appartenenti ad uno stessa funzionalità, andrebbero collocati in gruppi in macchine separate.

Infine, evitare l'installazione di pacchetti non strettamente necessari al funzionamento delle applicazioni nei container può rivelarsi un buon modo per evitare situazioni pericolose.

### Attacchi di Denial of Service

L'attacco di Denial-of-Service (DoS) è uno dei più noti attacchi tramite rete: un processo, o un gruppo di processi, prova a consumare totalmente le risorse del sistema al fine di interromperne il regolare esercizio.

In un'architettura a containers, in cui tutti i containers condividono le risorse del kernel, un DoS può verificarsi quando un container exploita l'accesso ad una risorsa, privandone l'accesso a tutti gli altri containers.

Per contrastare questo tipo di attacchi, si possono adottare soluzioni a livello del sistema operativo: isolare i file system, isolare lo spazio dei processi, ovvero la rete ed i dispositivi utilizzabili [5] . Uno strumento per ottenere configurazioni di questo genere è *cgroups* che consente di impostare e gestire limiti di accesso alle risorse da parte

di processi e container Docker. Così facendo, non solo ci si assicura che ciascun container abbia un'appropriata fetta di risorse ma si evita anche la monopolizzazione delle risorse da parte di uno o più containers.

### Container Breakouts

In questa tipologia di attacco, l'attacker prova a "evadere il proprio container", al fine di ottenere l'accesso alla macchina host e/o agli altri containers in esecuzione.

La funzione *open_by_handle_at()* consente ad un processo di accedere ad un file system. Per utilizzarla bisogna fornire la capability *CAP_DAC_READ_SEARCH*. Un superuser avrà questa capability di default e ciò significa che un attacker è autorizzato a bypassare i vincoli di *simfs* per accedere ai file su di un file system della macchina host. Questa vulnerabilità è stata fixata Docker 0.12, quindi il principale suggerimento è quello di aggiornare la versione di Docker.

Nel caos in cui non sia possibile aggiornare Docker, per mitigare questa vulnerabilità si può impostare il file system del container in sola lettura.

### Immagini avvelenate

All'interno delle immagini Docker è possibile iniettare software malevolo, come virus e trojan.

Docker classifica le immagini scaricate come "verificate" se la fingerprint dell'immagine scaricata corrisponde a quella specificata nel manifest, firmato, dell'immagine stessa. Sotto questo scenario, un attacker potrebbe trasmettere un'immagine avvelenata purché affiancata da un manifest firmato [29] .

In Docker, le immagini sono scaricate da un server HTTPS. Tuttavia queste stesse immagini passano attraverso un'insicura pipeline:

$$[decompressione] \longrightarrow [checksum] \longrightarrow [scompattazioneimmagine]$$

In Docker questa pipeline è poco protetta e, pertanto, gli utenti dovrebbero scaricare solamente immagini autenticate e di cui ci si può fidare.

Un ulteriore suggerimento potrebbe essere quello di bloccare l'accesso a *index.docker.io* al fine di scaricare le immagini solo manualmente, prima di importarle nel proprio ambiente Docker.

### Segreti compromessi

La compromissione di segreti, come ad esempio chiavi per l'utilizzo di API oppure password di database, può compromettere la sicurezza di interi sistemi. Per proteggere Docker da questa tipologia di attacchi è necessario impostare i file system dei container in sola lettura. Inoltre, per la condivisione di segreti l'utilizzo di variabili d'ambiente, seppur largamente diffusa come strategia, è fortemente sconsigliato [39].

### Man in the Middle

Negli attacchi di tipo Man in the Middle (MitM) un utente malintenzionato si inserisce in un canale di comunicazione tra due (o più) entità che si fidano tra loro. Pertanto, in questo modo, un attaccante riesce a rubare, o addirittura, modificare, informazioni sensibili scambiate tra le due (o più) entità fidate.


Figura 3.6: Man in the Middle

Per evitare questi attacchi nei containers Docker, l'isolamento della rete è la miglior strategia. Una oculata configurazione di rete si rivela essere la chiave per evitare che un utente malintenzionato possa sniffare, o manipolare, il traffico che viaggi sulla rete tra container, o sulla rete dell'host stesso [5].

Per l'isolamento è possibile avvalersi di strumenti terzi che implementano, ad esempio, delle VPN crittografate. Proprio in questo

senso, Docker offre la possibilità di incapsulare al suo interno un server OpenVPN per la gestione delle reti private all'interno dell'ambiente a containers.

### ARP spoofing

Lo spoofing ARP (Address Resolution Protocol) è un attacco in cui un attacker invia un messaggio ARP fasullo in LAN al fine di associare il proprio indirizzo MAC all'indirizzo IP di un sistema target. In questo modo, l'attacker riceverà tutti i dati scambiato con il sistema target.

Se un attacker riesce ad ottenere un privilegio di questo tipo, sarà in grado di ottenere e manipolare informazioni sensibili, come ad esempio password o dati sensibili di utenti che utilizzano un'applicazione. Si immagini lo scambio dati tra una web application ed un container di database. In aggiunta, l'attacker potrebbe iniettare del payload malevolo nella rete.

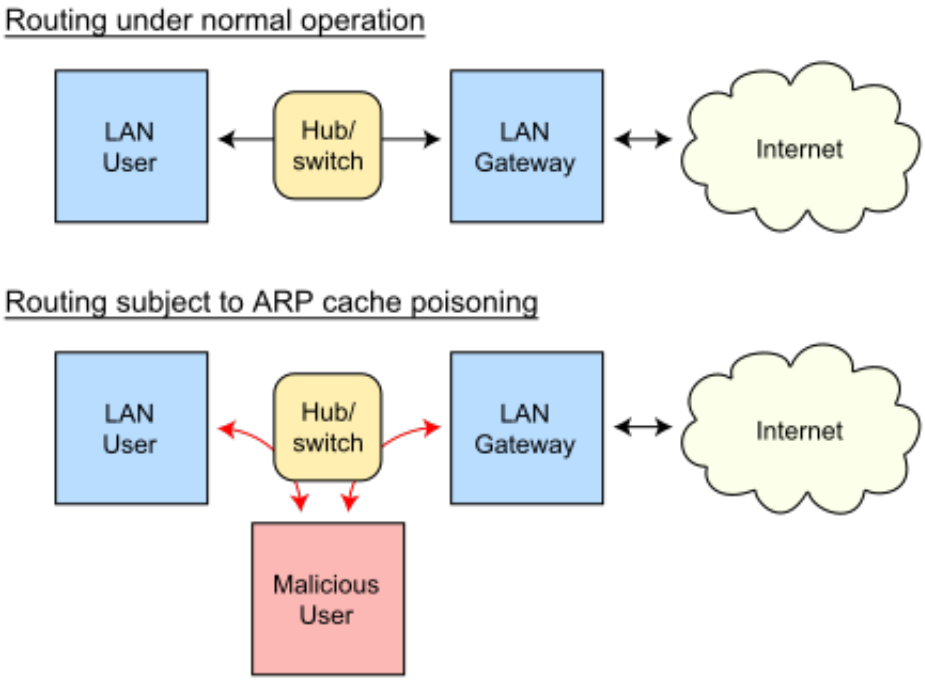


Figura 3.7: ARP Spoofing

Per mitigarlo, uno dei modi più efficaci è quello di avviare il container senza la capability *NET_RAW*, cosicché i programmi nei contianer non possono creare socket *PF_PACKET* e, di conseguenza, non potranno tentare attacchi di ARP spoofing.

Un'altra strategia per la mitigazione è quella di utilizza le *ebtables* per filtrare le frame Ethernet, in modo da catturare ed identificare i pacchetti ARP malevoli [23].

Adottando una di queste configurazioni, un attacker non avrà la possibilità di sferrare un attacco di ARP spoofing, o comunque ciò risulterà più complicato.

## 3.3 Estensione al formalismo MACM per container

L'estensione al formalismo MACM (Multi-cloud Application Composition Model) [7], che descrive i componenti logici che costituiscono il sistema e le loro relazioni mediante un grafo, vede l'introduzione di un nodo, Container as a Service, che modella il framework offerto da un provider per l'orchestrazione ed il deploy dei container. Quando si tratta di sistemi sicuri, con l'emergere dei *Container as a Service* (CaaS), è necessario evitare l'esecuzione di containers su host non attendibili [30]. Numerosi attacchi, inclusi attacchi sia attivi che passivi, possono essere lanciati da host semi-attendibili e non attendibili contro i containers in esecuzione al loro interno. Esempi di attacchi passivi includono la profilazione delle attività dei container (e delle applicazioni in esecuzione al loro interno) e l'accesso non autorizzato ai dati dei container stessi. Attacchi attivi possono essere ancora più dannosi poiché potrebbero far variare il comportamento delle applicazioni [1]. Particolare attenzione va posta anche ai container già offerti dal provider che potrebbero essere software malevolo. Per queste ragioni, le tipologie di nodi del sistema sono:

- **Software as a Service**, modella il software in esecuzione (containerizzato)
- **Container as a Service**, modella il gestore dei container

- **Infrastructure as a Service**, modella una virtual machine
- **Cluster**, modella il cluster che hosta i container
- **CSP**, modella il provider che offre la IaaS e/o CaaS

Mentre gli archi:

- Un nodo CSP può fornire (*provides*) uno o più IaaS oppure un CaaS.
- Un nodo CaaS può gestire (*manages*) uno o più Cluster.
- Un nodo Cluster può hostare (*hosts*) uno o più SaaS
- Un nodo IaaS può hostare (*hosts*) uno o più SaaS
- Un nodo SaaS può utilizzare (*uses*) altri SaaS.

In aggiunta alle informazioni in merito all'architettura dell'applicazione, il formalismo MACM consente di specificare anche informazioni di dettaglio per l'accesso al componente, come ad esempio l'indirizzo IP ed il porto per accedere al servizio.

In Figura 3.8 è presentato un esempio di sistema modellato tramite il formalismo proposto. Il sistema è composto da un'applicazione "APP" che utilizza un database "DB", entrambi in esecuzione in container, per mezzo di un orchestratore direttamente offerto da un generico provider.

In Figura 3.9 è presentato lo stesso esempio ma questa volta l'orchestratore è locato su di una macchina dedicata e non offerto direttamente dal CSP.

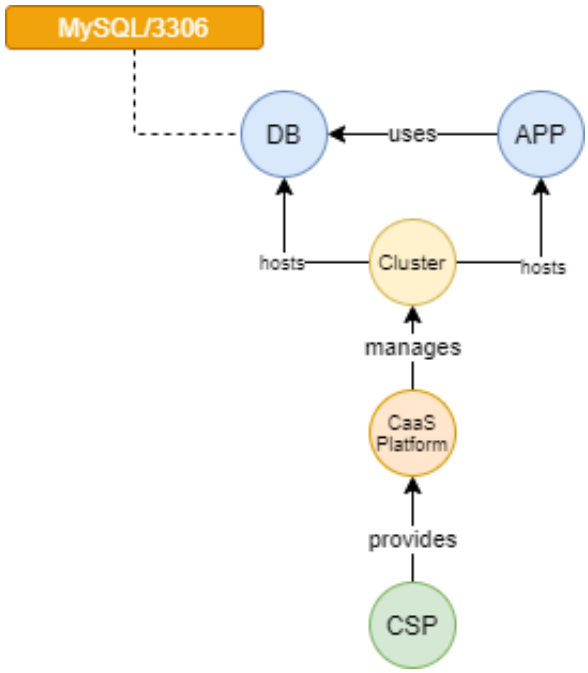


Figura 3.8: Esempio modellazione MACM CaaS

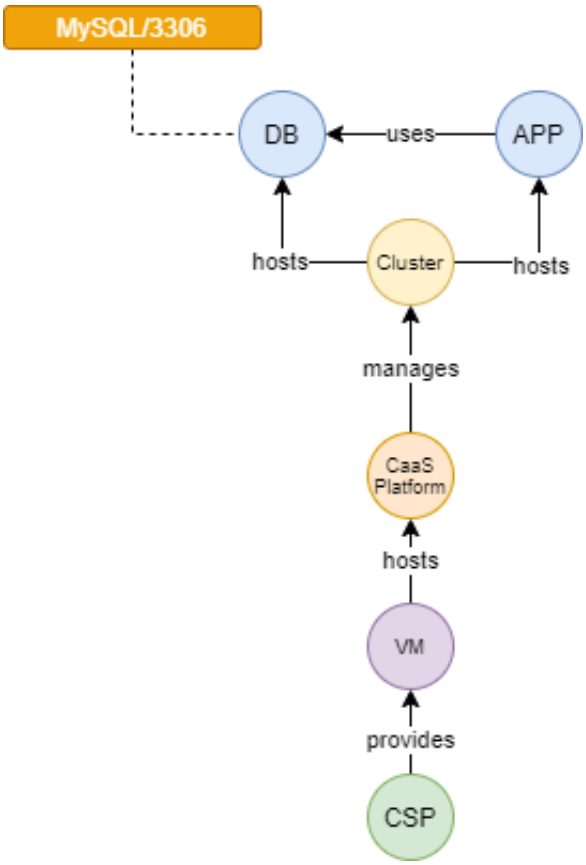


Figura 3.9: Esempio modellazione MACM con CaaS su VM

# Capitolo 4

# Realizzazione di uno strumento di testing automatico

Al fine di mostrare la fattibilità della metodologia proposta, è stato progettato e realizzato uno strumento, integrabile in un workflow di CI/CD, che implementa la metodologia proposta.

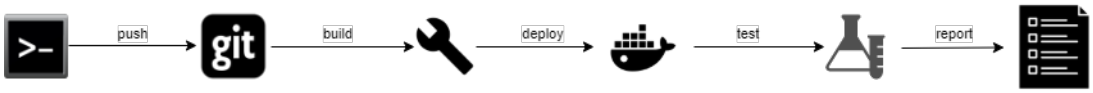


Figura 4.1: Workflow di sviluppo

Lo strumento, automatico, è eseguibile al verificarsi di un dato evento, come ad esempio al deploy del sistema in un ambiente a con-

tainer di sviluppo in seguito ad un *push* sul repository git del sistema software di interesse.

## 4.1 Workflow implementato

Lo strumento implementa il seguente flusso:

1. **Scansione del sistema**, tramite strumenti di security scanning
2. **Generazione del modello** di sicurezza del sistema
3. **Esecuzione test** di sicurezza
4. **Generazione report** in formato HTML

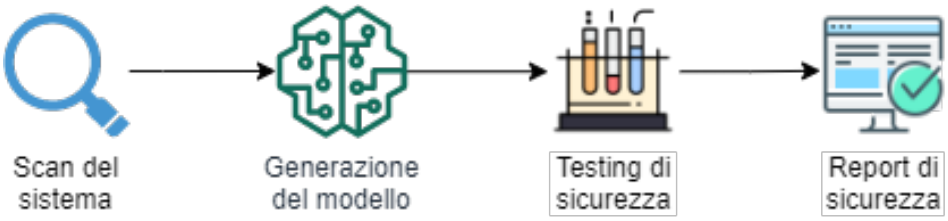


Figura 4.2: Workflow strumento

### 4.1.1 Scansione del sistema

La fase iniziale di scansione del sistema prevede l'esecuzione di differenti strumenti di security scanning. Nel caso specifico di esempio, sono stati integrati:

- **OWASP ZAP**, uno scanner di sicurezza per applicazioni Web. Scansiona gli endpoint dell'applicazione web in cerca di vulnerabilità note.

- **Anchore Container Analysis**, uno scanner di immagini di container. Scansione l'immagine in cerca di vulnerabilità note a livello del sistema, e dei pacchetti utilizzati, ed a livello dell'applicazione.

- **JoomScan**, uno scanner di vulnerabilità note per il Content Management System Joomla!

Tutti gli strumenti offrono una modalità di utilizzo *headless* che rende semplice l'integrazione in processi automatici di CI/CD.

L'output di questi strumenti è dato in pasto ad un motore che lo analizza per generare il modello di sicurezza del sistema.

### 4.1.2 Generazione del modello di sicurezza

L'output degli strumenti di scansione è utilizzato per la generazione del modello di sicurezza, avvalendosi in aggiunta delle informazioni presenti nei database MITRE *Common Weakness Enumeration (CWE)* e *Common Vulnerabilities and Exposures (CVE)*, per ciascuna weakness rilevata mappandola con il relativo controllo di sicurezza NIST [19] .

Più nello specifico, il motore sfrutta i database MITRE *Common Weakness Enumeration (CWE)* e *Common Vulnerabilities and Expo-*

*sures (CVE)* per popolare il modello con le informazioni correlate alla vulnerabilità rilevata in termini di attacchi noti, contromisure da adottare, livello di rischio, etc. Specifica, inoltre, anche la particolare istanza del sistema da sollecitare quando si vuole tentare un attacco.

Per semplificare la lettura e la comprensione del lavoro, in Figura 4.3 è nuovamente riportata la struttura del modello.

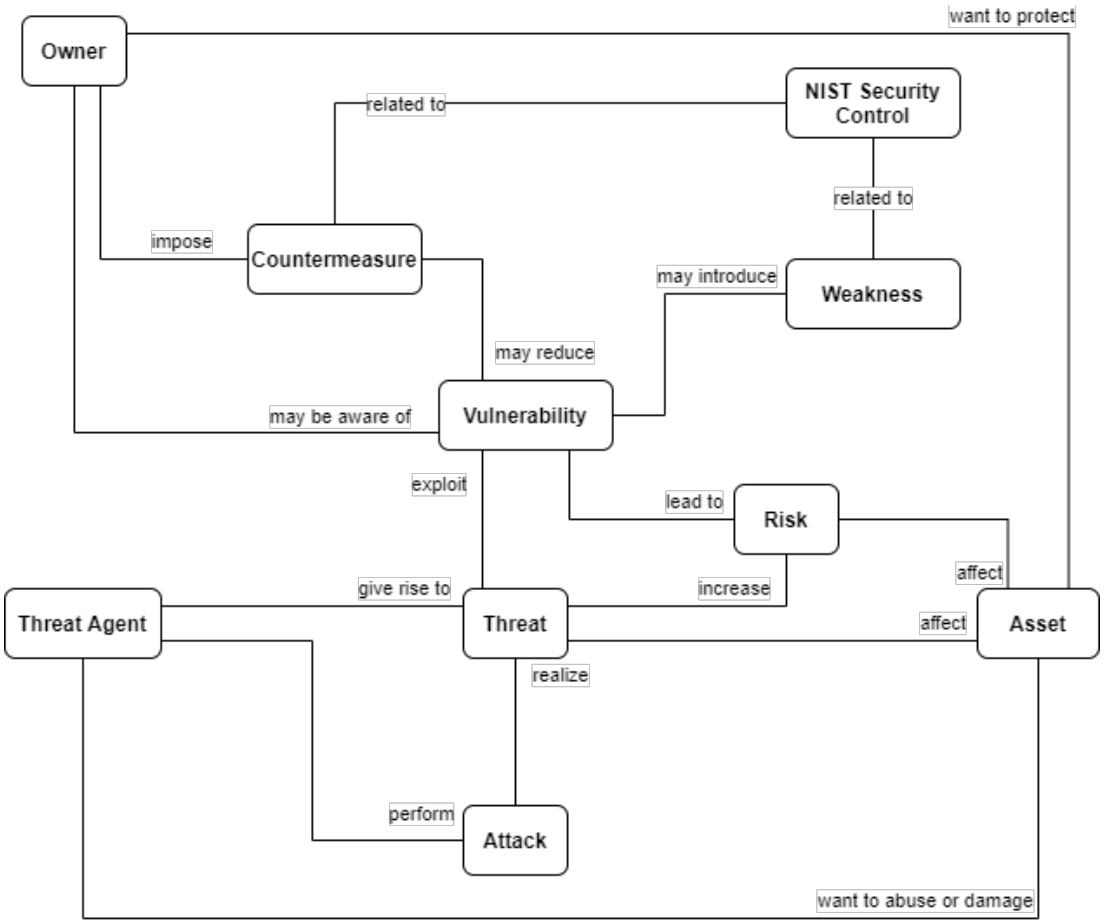


Figura 4.3: Modello della sicurezza

In aggiunta, per riconoscere univocamente una rilevazione, a ciascuna è assegnato un identificativo univoco.

### 4.1.3 Testing di sicurezza

Ottenuto il modello di sicurezza, per ogni rilevazione si avvia un test di sicurezza specifico, al fine di valutare l'effettiva exploitabilità delle weakness rilevate dagli strumenti di scansione.

La tipologia di attacco è stabilita dalle informazioni del modello e lo stesso vale per il punto di attacco: l'istanza rilevata dagli strumenti di scansione.

In questa fase si ipotizza di avere a disposizione uno o più strumenti terzi che implementano gli attacchi necessari a coprire tutte le rilevazioni. In questo caso specifico di esempio, è stata implementata una batteria di test automatici per attacchi noti, come SQL Injection ed Excavation (Collect Data from Common Resource Locations).

I problemi rilevati durante l'esecuzione immediatamente precedente dello strumento, e non più rilevati dalla scansione attuale, sono comunque sottoposti a testing al fine di confermare e/o smentire la loro risoluzione. In caso il sistema dovesse risultare comunque vulnerabile, il modello viene modificato aggiungendo nuovamente la rilevazione.

### 4.1.4 Generazione report

Al termine della fase di testing, viene valutato lo stato attuale del sistema anche in confronto a quanto appurato con le precedenti esecuzioni dello strumento. In questo senso, viene valutata la presenza di regressione nel sistema, segnalando opportunamente la versione del-

l'applicazione (ed il commit corrispondente) che ha, in precedenza, risolto il problema.

Per supportare la gestione del ciclo di evoluzione, il report mantiene un riferimento alla versione dell'applicazione (commit) su cui è stato eseguito il test, oltre che la data e l'ora di esecuzione di ciascun test.

Per ciascuna rilevazione si riportano:

- **Rischio**, rischio associato alla vulnerabilità secondo il database MITRE CWE.
- **Vulnerabilità**, nome della vulnerabilità rilevata
- **Weakness**, nome e descrizione della debolezza
- **Threat**, la potenziale minaccia
- **Instance**, la particolare istanza che ha generato la rilevazione
- **Asset**, gli asset a rischio
- **Countermeasure**, le contromisure da adottare per mitigare la rilevazione
- **Attack**, gli attacchi possibili per sollecitare la vulnerabilità
- **Probabilità di Exploit**, la probabilità che si verifichi un exploit, secondo il database MITRE CWE
- **Scan tool**, lo strumento di scansione che ha generato la rilevazione

- **Test result**, risultato del test di sicurezza eseguito. Determina la exploitability della weakness rilevata e può essere indeterminata (*unknowkn*) se non si ha a disposizione uno strumento che implementa l'attacco opportuno alla verifica

- **Timestamp**, marcatura temporale relativa all'esecuzione del test

- **NIST Security Control**, il controllo di sicurezza (standard NIST 800-53) relativo alla weakness rilevata

- **ID**, identificativo univoco assegnato alla vulnerabilità rilevata sulla specifica istanza

Con l'obiettivo di agevolare e semplificare un eventuale processo di assessment di sicurezza del sistema, viene riportato anche il controllo di sicurezza del NIST associato alla weakness. Il mapping CWE-NIST è fornito da MITRE [19].

Di seguito uno screenshot di un security report di esempio generato.

## Juice Shop v.11.1.3(initCommit) Security Report

| Medium | Hidden File Found |
|---|---|
| Weakness | CWE-538: Insertion of Sensitive Information into Externally-Accessible File or Directory. The product places sensitive information into files or directories that are accessible to actors who are allowed to have access to the files, but not to the sensitive information. |
| Threat | Read Files or Directories |
| Instance | GET http://localhost:3000/.svn/entries |
| Asset | Application |
| Countermeasure | Consider whether or not the component is actually required in production, if it isn't then disable it. If it is then ensure access to it requires appropriate authentication and authorization, or limit exposure to internal systems or specific source IPs, etc. |
| Attack | WSDL Scanning |
| Likelihood Of Exploit | Unknown |
| Scan tool | ZAP |
| Test result | FAIL |
| Date | 18/08/2020 22:07 UTC |
| ID | 830bd3dddccd118e62076935f7013684 |

| High | SQL Injection - SQLite |
|---|---|
| Weakness | CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. |
| Threat | Read Application Data; Bypass Protection Mechanism; Modify Application Data |
| Instance | GET http://localhost:3000/rest/products/search?q= |
| Asset | Application |
| Countermeasure | Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side.If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.If database Stored Procedures can be used, use them.Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!Do not create dynamic SQL queries using simple string concatenation.Escape all data received from the client.Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.Apply the privilege of least privilege by using the least privileged database user possible.In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.Grant the minimum database access that is necessary for the application. |
| | Command Line Execution through SQL Injection; Object Relational Mapping Injection; SQL Injection through SOAP Parameter Tampering; |

Figura 4.4: Esempio report di sicurezza

## 4.2 Architettura dello strumento

Lo strumento realizzato prevede l'utilizzo di diversi strumenti terzi per la scansione del sistema in cerca di vulnerabilità e per il testing di sicurezza.

Viene proposta una versione cui è integrato uno strumento di scansione di immagini Docker (Anchore [2]), uno di scansione di applicazioni web (OWASP ZAP [25]) ed uno scanner di vulnerabilità per siti web realizzati con Joomla! (JoomScan [28]).

Per quanto riguarda il testing di sicurezza, invece, si ipotizza di avere a disposizione uno strumento (o più) che implementa gli attacchi proposti dal modello generato, per verificare l'exploitabilità di ciascuna weakness rilevata. In questo caso, non essendo in possesso di uno strumento tale, è stata realizzata una piccola batteria di test automatici che implementa alcuni attacchi noti.

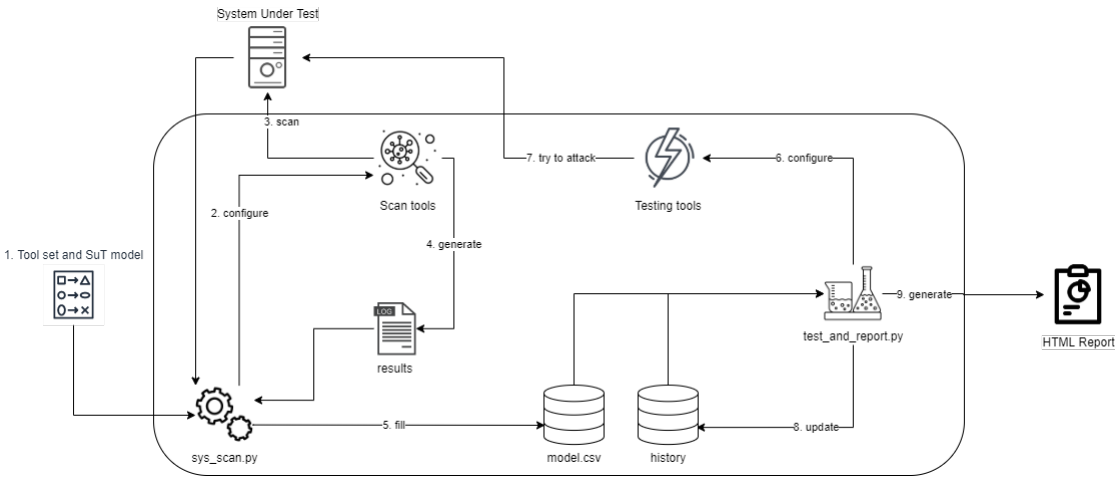


Figura 4.5: Vista architetturale dello strumento

L'architettura a pipeline descritta in Figura 4.5, prevede un primo modulo che orchestra gli strumenti di scansione ed elabora i loro output: "*sys_scan.py*". Nel dettaglio, si occupa di orchestrare gli strumenti richiesti per lo scan del System Under Test, uniformando i risultati ottenuti e generando un unico modello di sicurezza dello stato attuale del sistema.

Il modello generato, assieme allo storico delle precedenti esecuzioni dello strumento (*history*), è inviato in input al modulo "*test_and_report.py*" che si occupa di analizzare i dati ed orchestrare opportunamente gli strumenti di testing automatico. La configurazione dei testing tools, prevede l'analisi di ciascun record del modello, al fine di identificare la tipologia da attacco da sferrare e l'istanza del sistema da sollecitare. Con queste informazioni, si configura lo strumento di testing per sferrare l'attacco proposto al fine di valutare l'exploitabilità della weakness rilevata.

L'esito del processo è infine serializzato in un file HTML human-readable.

### 4.2.1 File prodotti

Lo strumento per ogni esecuzione genera diversi file e directory. La fase di scansione, governata dal modulo *sys_scan.py*, genera:

- **model.csv**, modello dello stato attuale del sistema,

- File di output dei vari strumenti utilizzati. Questi sono utilizzati dal modulo per uniformare i risultati secondo il modello di sicurezza di riferimento e restano sul disco.

La fase di testing e report, governata dal modulo *test_and_report.py*, genera:

- **report.html**, file di report HTML human-readable (vedi Paragrafo 4.1.4 )
- **historyReport.html**, report storico del sistema (vedi Paragrafo 4.4.2 )
- **last-model.csv**, il modello dello stato attuale del sistema viene così rinominato una volta generato il report, al termine dei test. E' utilizzato per calcolare la differenza tra lo stato attuale e quello precedente, alla prossima esecuzione dello strumento.
- **.model-DDMMAAAAHHMM.csv**, il precedente *last-model.csv* ovvero il penultimo stato del sistema (utile per fare una differenza tra lo stato attuale e quello precedente) non viene eliminato ma archiviato in questo modo. DDMMAAAAHHMM è una stringa che rappresenta la marcatura temporale cui il file è archiviato.
- **history**/, una directory contenente lo storico di tutte le rilevazioni

- history/**history.csv**, mantiene lo storico delle statistiche, in funzione della versione/commit dell'applicazione.

## 4.3 Sviluppo

Lo strumento è scritto in Python (3.6.9), necessita di una connessione ad internet per consultare i database esterni MITRE e fa uso di alcune librerie esterne:

- *requests_html*, accesso ai database MITRE *Common Weakness Enumeration (CWE)* e *Common Vulnerabilities and Exposures (CVE)*
- *json*, elaborazione di output in formato JSON degli strumenti
- *csv*, lettura/scrittura di file in formato CSV
- Altre librerie standard come *os*, *sys* e *datetime* per l'accesso al disco, ai parametri passati in ingresso ed all'orario della macchina

### 4.3.1 Struttura componenti e relazioni

La struttura dei componenti e le loro relazioni è raffigurata in Figura 4.6.

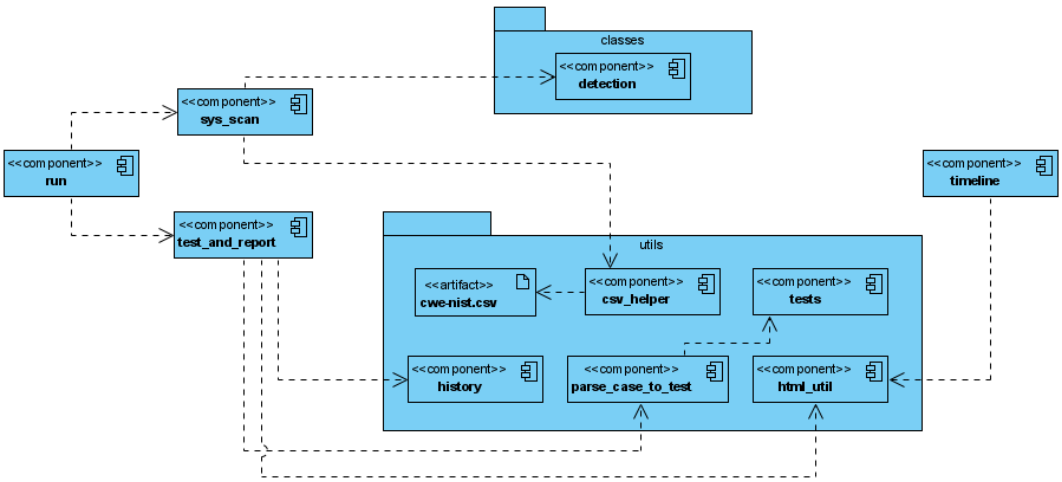


Figura 4.6: Componenti e le loro relazioni

Il package *utils* contiene:

- *csv_helper*, helper per la scrittura su csv
- *html_util*, contiene i template HTML utilizzati per i report
- *history*, helper per la gestione dello storico
- *parse_case_to_test*, contiene le funzioni di utilità per analizzare un record del modello, per stabilire quale test eseguire
- *tests*, piccola batteria di test di esempio
- *cwe-nist.csv*, il file che mappa le weakness ai controlli NIST 800-53. Il mapping CWE-NIST è fornito da MITRE [19].

Il package *classes* contiene:

- *detection*, la classe *Detection* definita all'interno rappresenta un record nel modello di sicurezza

Il componente *run*, il cui utilizzo è dettagliato al Paragrafo 4.5, avvia i componenti *sys_scan* e *test_and_report* che si occupano, rispettivamente, della fase di scansione e testing del sistema.

*sys_scan* avvia gli strumenti di scansione (richiesti in input - vedi Paragrafo 4.5) e per ciascuna rilevazione costruisce un oggetto *Detection* da scrivere nel modello tramite *csv_helper*.

*test_and_report* analizza la storia delle scansioni (tramite *history*) e passa ciascuna *Detection* a *parse_case_to_test* che la analizza al fine di stabilire quale test avviare tra quelli disponibili nel componente *tests*. L'esito è poi restituito a *test_and_report* per la scrittura nel report in formato HTML il cui template è letto da *html_util*.

Il componente *timeline* si occupa della generazione on-demand delle sequenze temporali che mostrano l'evoluzione nel tempo di una rilevazione. Fa uso di *html_util* per la generazione dell'output in formato HTML.

### 4.3.2 Estendere lo strumento

#### Aggiunta di ulteriori strumenti di scansione

Per poter estendere lo strumento, con strumenti di scansione aggiuntivi, occorre modificare il file *sys_scan.py* in questo modo:

1. Aggiungere il nome simbolico dello strumento alla lista degli strumenti supportati:

```
AVAILABLE_TOOLS = ['anchore', 'zap', 'joomscan',
   'nuovotool']
```

2. Avviare lo strumento di scansione:

```
os.system('nuovotool ...')
```

3. Per ciascuna rilevazione, definire una nuova istanza della classe *Detection* popolata con le informazioni richieste. Se non sono note, utilizzare *None* cosicché lo strumento possa recuperarle, se possibile, dal database MITRE CWE.

```
curr_detection = Detection(
   cwe_url, # URL Weakness da database MITRE CWE
   risk, # Rischio associato, se None sara' letto
      dal database
   threats, # Minacce associate, se None sara'
      letto dal database
   vulnerability, # Vulnerabilita' associata, se
      None sara' letta dal database
   weakness, # Weakness rilevata, se None sara'
      letta dal database
   instance, # Istanza cui e' avvenuta la
      rilevazione
   assets, # Asset coinvolti
   countermeasure, # Contromisura proposta, se None
      sara' letta dal database
   attacks, # Attacco da utilizzare per verificare
```

```
        l'exploitabilita', se None sara' letto dal
        database
    likelihood_of_exploit, # Probabilita' di
        exploit, se None sara' letto dal database
    scan_tool # Strumento che ha generato la
        rilevazione
)
```

Se l'url della weakness nel database MISTE CWE non è noto, è possibile recuperarlo tramite l'ID della weakness e/o della vulnerabilità, utilizzando la funzione di utilità *retrieveCWEURLFromCVE*.

4. Scrivere le informazioni di ciascuna istanza di *Detection* nel file csv di modello:

```
csv_helper.writeRow(csv_model_writer,
    curr_detection)
```

### Modifica logica di selezione dei test

Per ciascuna rilevazione, è invocata la funzione *parse* del file *utils/parse_case_to_test.py*. Qui risiede la logica di scelta ed avvio dei test, oltre che di manipolazione del risultato degli stessi.

### Aggiunta di ulteriori pentest

Il modulo che si occupa di avviare i test è *utils/tests.py*, in cui ogni test è definito come una funzione. Per l'aggiunta di ulteriori test è possibile definire nuove funzioni al modulo e richiamarle, all'occorrenza, dalla funzione *parse* di *utils/parse_case_to_test.py*.

E' inoltre possibile, oltre che consigliato, interfacciare lo strumento con tool terzi di testing. Per farlo basta, ancora una volta, definire una nuova funzione che avvia lo strumento di testing opportunamente configurato.

Un test restituisce:

- **True**, test terminato con successo: exploitabilità non rilevata
- **False**, test fallito: exploitabilità rilevata

Un esempio:

```
def myPentest(instance):
   os.system('pentestingtool instance')
   * manipolazione output *
   return True
```

## 4.4 Funzionalità aggiuntive

Lo strumento propone, in aggiunta, delle funzionalità extra volte a semplificare ed automatizzare, oltre l'analisi, anche l'assessment di sicurezza delle applicazioni.

### 4.4.1 Evoluzione di una rilevazione

Per ciascuna rilevazione riportata nel report generato (Paragrafo 4.1.4), è possibile richiedere la generazione di una sequenza temporale che ne mostra l'evoluzione nel tempo, in funzione della versione dell'applicazione e del relativo commit. Ciò risulta utile quando si vuole analizzare la presenza di una weakness (in termini di exploitability, in relazione all'esito del pentest eseguito) con l'evolversi del software.

Di seguito uno screenshot di una timeline generata per una rilevazione di esempio.

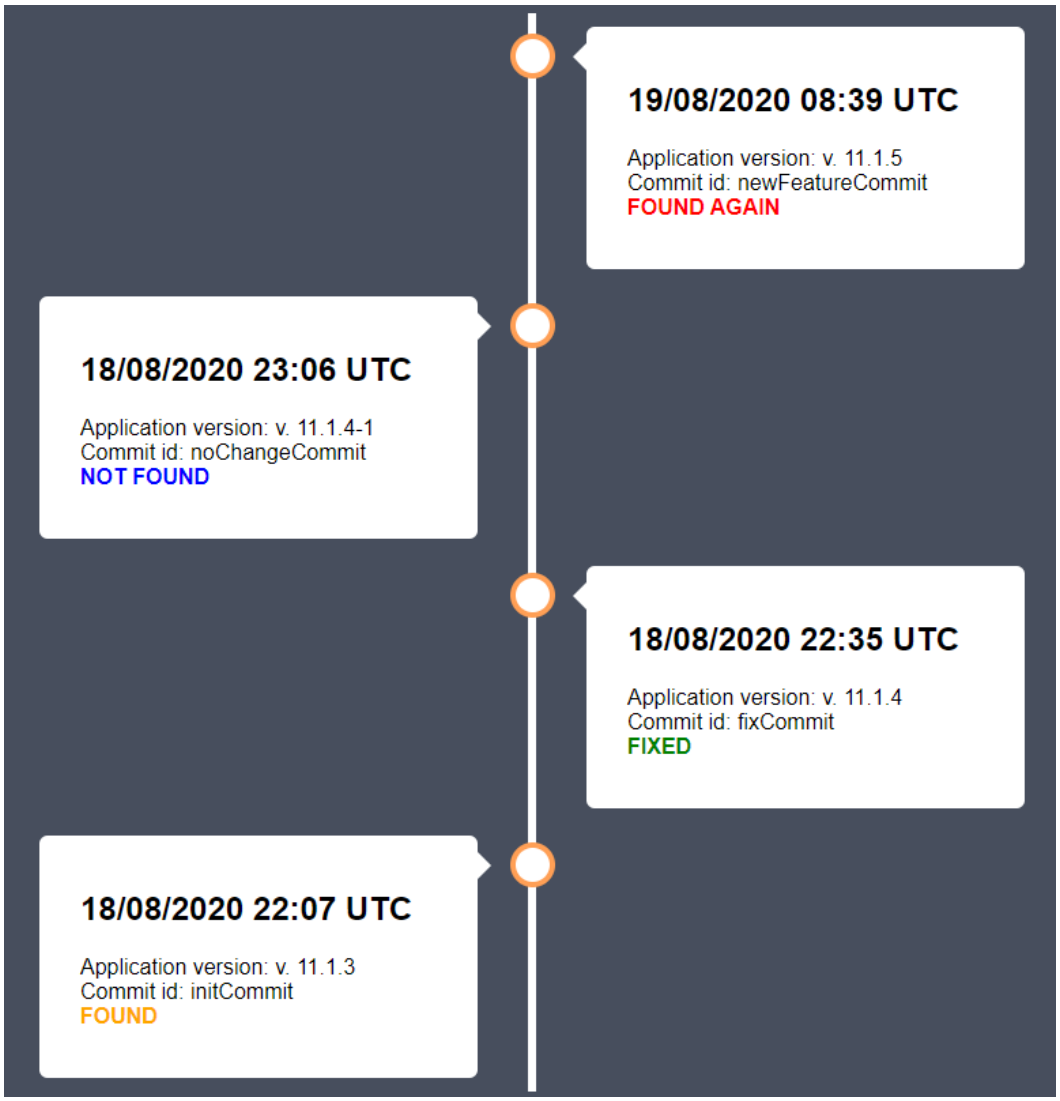


Figura 4.7: Esempio timeline rilevazione

### 4.4.2 Evoluzione dello stato complessivo

A valle di ciascuna esecuzione dello strumento, viene generato un ulteriore report che visualizza l'andamento dello stato complessivo del sistema, riportando la variazione del numero di rilevazioni, suddivise per livello di rischio, nel tempo.

Di seguito uno screenshot di uno storico di esempio generato.


Figura 4.8: Esempio timeline storico

### 4.4.3 Statistiche

Per ciascun report viene riportato un piccolo riquadro statistico, utile ai fini del security assessment, che indica, per ciascun livello di rischio, quante weakness sono state rilevate ed exploitate (o potenzialmente exploitabili se non hanno a disposizione strumenti che implementato il test necessario) si sono verificate ed i controlli di sicurezza NIST 800-53 che falliscono.

| Risk | Number of occurrences |
|---|---|
| medium | 182 |
| low | 108 |
| high | 5 |
| critical | 7 |
| **Total:** | 302 |

Figura 4.9: Esempio statistiche report di sicurezza

| Failing NIST Security Controls |
|---|
| SC-13, SI-11, SC-8, AC-3, SI-10 |

Figura 4.10: Esempio statistiche report di sicurezza NIST 800-53

Queste informazioni sono estremamente utili in fase di valutazione del sistema in termini di sicurezza.

## 4.5 Utilizzo dello strumento

Per l'utilizzo headless dello strumento, facilmente integrabile in contesti di CI/CD, è stato realizzato uno script *run.sh*. Il suo utilizzo prevede:

```
sh run.sh APP_NAME APP_VERSION COMMIT_ID IMAGE_NAME
   IMAGE_TAG APP_URI [tool1 tool2 ... toolN]
```

- **APP_NAME**, nome applicazione
- **APP_VERSION**, versione applicazione
- **COMMIT_ID**, id commit di riferimento
- **IMAGE_NAME**, nome immagine Docker da analizzare
- **IMAGE_TAG**, tag immagine Docker da analizzare
- **APP_URI**, identificativo della risorsa che rappresenta il System Under Test (ad esempio l'URL cui è deployata l'applicazione)
- **tool**, la lista dei tool da utilizzare in fase di scansione

Esempio d'uso:

```
sh run.sh "Mia Applicazione" 0.1-beta myCommitId
   myappimage 0.0.1 http:\\localhost:3000 anchore zap
```

### 4.5.1 Utilizzo avanzato dello strumento

Un utilizzo avanzato dei componenti dello strumento consente di separare le fasi di scansione e test. E' ad esempio possibile avviare la sola fase di scansione, ed aggiornare dunque il modello corrente, senza eseguire la fase di test né ottenere il report HTML. Per fare ciò si può utilizzare il modulo di *sys_scan.py* in maniera indipendente, con gli stessi parametri della versione completa:

```
python sys_scan.py APP_NAME APP_VERSION COMMIT_ID
   IMAGE_NAME IMAGE_TAG APP_URI [tool1 tool2 ... toolN]
```

## 4.6 GitHub Action

Le GitHub Actions consentono la definizione di job event-driven, ovvero è possibile eseguire una serie di comandi al verificarsi di specifici eventi.

Al fine di integrare lo strumento in un workflow di CI/CD, è stata realizzata una Github Action di esempio dedicata che avvia lo strumento automaticamente al verificarsi di un evento di push.

Il flusso eseguito ad ogni evento è il seguente:

1. *Set up job*, configura la macchina GitHub per l'esecuzione della Action

2. *Run actions/checkout@v1*, esegue il check-out del repository così che la Action vi possa accedere
3. *Set up environment*, configura la macchina con i software necessari all'esecuzione dello strumento
4. *Install dependencies*, installa le dipendenze software come i pacchetti Python richiesti
5. *Execute*, avvia lo strumento
6. *Run actions/upload-artifact@v2*, queste fasi servono per rendere disponibili gli artefatti generati come ad esempio il report in formato HTML

I parametri da fornire in input (vedi Paragrafo 4.5) sono elaborati automaticamente:

- **APP_NAME**, nome del repository GitHub
- **APP_VERSION**, numero univoco che identifica l'esecuzione
- **COMMIT_ID**, id commit GitHub che ha triggerato l'evento
- **IMAGE_NAME**, nome immagine Docker ottenuto dal processo di building
- **IMAGE_TAG**, tag immagine Docker ottenuto dal processo di building

- **APP_URI**, identificativo della risorsa che rappresenta il System Under Test (ad esempio l'URL cui è deployata l'applicazione) da specificare manualmente a monte della prima esecuzione della Action

- **tool**, la lista dei tool da utilizzare in fase di scansione è da specificare manualmente a monte della prima esecuzione della Action

L'elaborazione proposta dei parametri può essere variata utilizzando logiche custom per la loro definizione.

Di seguito si riporta uno screenshot della GitHub Action in esecuzione.

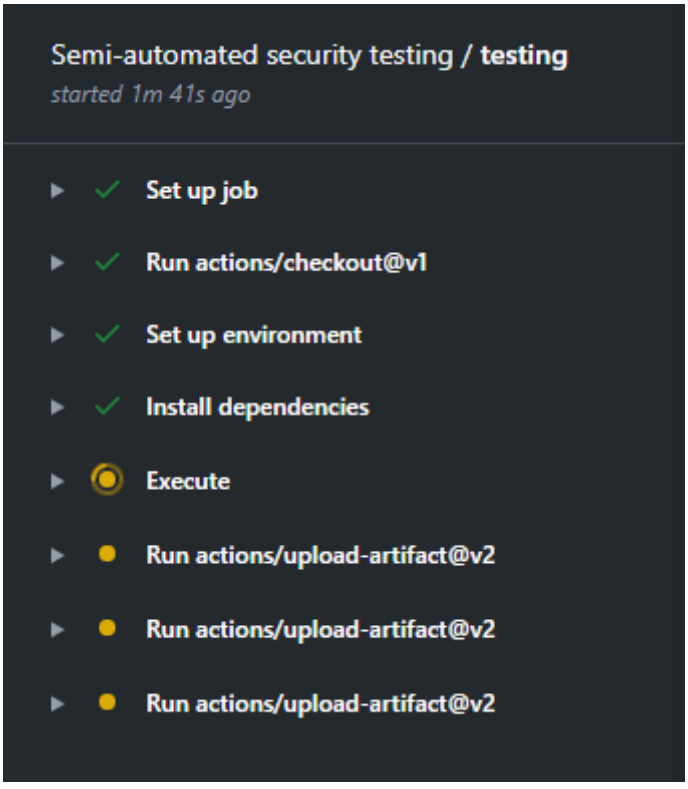


Figura 4.11: Esecuzione GitHub Action post push

Al termine dell'esecuzione, nella pagina relativa alla GitHub Action, è possibile visualizzare e scaricare gli artefatti generati.

Nello specifico ritroviamo:

- **report**, report HTML prodotto dallo strumento
- **last-model**, modello di sicurezza del sistema popolato dallo strumento, in formato CSV
- **historyReport**, report con lo storico delle statistiche
- **history**, cartella history prodotta dallo strumento

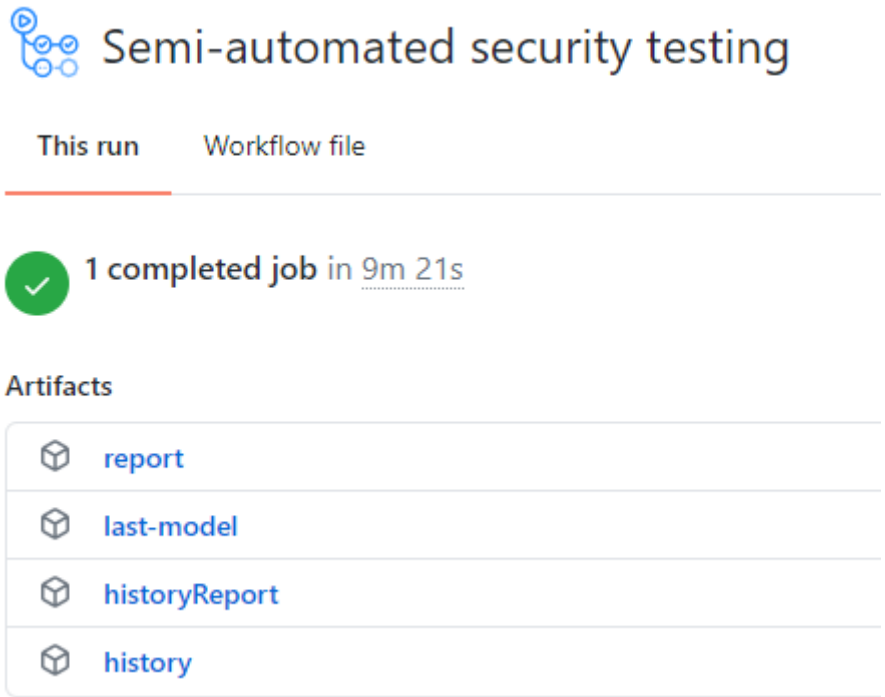


Figura 4.12: Artefatti generati GitHub Action

# Capitolo 5

# Caso di studio: Juice Shop

Figura 5.1: Juice Shop

Juice Shop [11] è un'applicazione web moderna, volutamente insicura. E' spesso utilizzata come applicazione di esempio per le esercitazioni nel campo della security, e come banco di prova per strumenti che analizzano la sicurezza delle applicazioni, specialmente quelle web.

Juice Shop è un'applicazione in esecuzione su Node.js ed utilizza i framework Express ed Angular. L'applicazione presenta un vasto numero di sfide, di diversa difficoltà, da dover risolvere: l'utente è

invitato a scoprire le vulnerabilità del sistema e ad escogitare un modo per exploitarle. Comprende le vulnerabilità presenti nella *OWASP Top Ten*, assieme a tante altre, per un totale di 95 sfide.

L'applicazione, opportunamente dockerizzata, viene utilizzata in questo lavoro come caso di studio della metodologia proposta per il testing di sicurezza.

L'obiettivo è quello di mostrare come lo strumento possa essere un supporto, automatico, per semplificare la gestione della sicurezza di un sistema in sviluppo. Pertanto, si simulano diversi cicli di evoluzione al fine di denotare come lo strumento sia in grado di rilevare automaticamente vulnerabilità di sicurezza (e proporre delle correzioni) e monitorare la loro evoluzione, oltre che lo stato generale del sistema, dal punto di vista della sicurezza, nel tempo.

Trattandosi di una applicazione web, come strumenti per la fase di scansione del sistema vengono utilizzati *Anchore* e *OWASP ZAP*.

## 5.1 Configurazione ambiente

Per poter riprodurre l'esperimento occorre scaricare e installare i seguenti software:

- Docker 19.03.12
- Python 3.6.9
- Immagine Docker owasp/zap2docker-stable:2.9.0

- Immagine Docker anchore/inline-scan:v0.8.1
- Immagine Docker Juice Shop bkimminich/juice-shop:v11.1.3
- Package Python requests-html 0.10.0

## 5.2 Descrizione del sistema mediante il formalismo MACM

L'applicazione è in esecuzione all'interno di un container, in un ambiente Docker installato su di una macchina virtuale gestita da una piattaforma di virtualizzazione (Proxmox) ospitata in un'infrastruttura cloud, presso l'Università della Campania "Luigi Vanvitelli" (Aversa, Caserta, Italia).

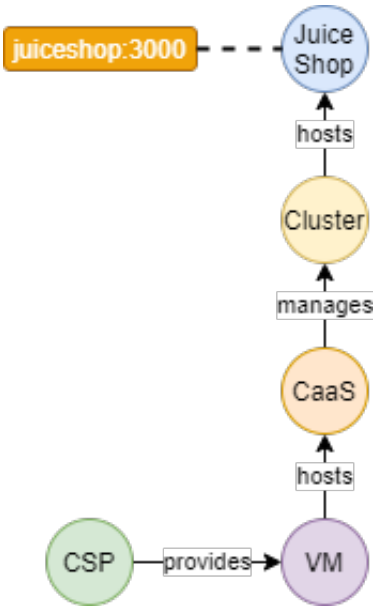


Figura 5.2: OWASP Juice Shop MACM

## 5.3 Configurazione dello strumento

La configurazione prevede principalmente la scelta degli strumenti di scansione da utilizzare, tra quelli integrati. Trattandosi di una applicazione web, come strumenti per la fase di scansione del sistema vengono utilizzati *Anchore* e *OWASP ZAP*.

Inoltre, ad ogni esecuzione (manuale o automatizzata) occorre passare in input la versione dell'applicazione e l'ID del commit di riferimento.

Il comando sarà dunque:

```
sh run.sh "Juice Shop" [versione applicazione] [id
   commit] myjuice [versione immagine]
   http:\\localhost:3000 anchore zap
```

## 5.4 Analisi cicli di evoluzione del sistema

Una prima esecuzione dello strumento è effettuata al verificarsi del push della versione di Juice Shop v11.1.3, presente sul repository ufficiale: https://github.com/bkimminich/juice-shop/.
Si tratta dunque della versione originale sistema, offerta dagli sviluppatori. Lo strumento è avviato con il seguente comando:

```
sh run.sh "Juice Shop" 11.1.3 initCommitId myjuice
   11.1.3 http:\\localhost:3000 anchore zap
```

Ciò sta ad indicare, così come specificato al Paragrafo 4.5, l'avvio dello strumento con i seguenti parametri:

- Juice Shop, nome dell'applicazione
- 11.1.3, versione dell'applicazione
- initCommitId, id del commit di riferimento
- myjuice, nome immagine Docker
- 11.1.3, tag immagine Docker
- http://localhost:3000, home dell'applicazione web
- anchora zap, strumenti di scansione da utilizzare tra quelli disponibili (vedi Paragrafo 4.2)

Di seguito si riportano le statistiche di questa prima analisi assieme ad un estratto del report di sicurezza ottenuto.

**Juice Shop v. 11.1.3 (initCommitId) Security Report**

| Medium | CVE-2019-15847 |
|---|---|
| Weakness | CWE-331: Insufficient Entropy. The software uses an algorithm or scheme that produces insufficient entropy, leaving patterns or clusters of values that are more likely to occur than others. |
| Threat | Bypass Protection Mechanism |
| Instance | libgcc-9.2.0-r4 |
| Asset | Container |
| Countermeasure | Phase: Implementation Determine the necessary entropy to adequately provide for randomness and predictability. This can be achieved by increasing the number of bits of objects such as keys and seeds. |
| Attack | Session Credential Falsification through Prediction |
| Likelihood Of Exploit | Unknown |
| Scan tool | Anchore |
| Test result | UNKNOWN |
| Timestamp | 29/08/2020 09:28 UTC |
| ID | fd12571bf5d173bf07f3bcbf5764c5f9 |
| NIST Security Control | SC-13 |

| High | SQL Injection - SQLite |
|---|---|
| Weakness | CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. |
| Threat | Read Application Data; Bypass Protection Mechanism; Modify Application Data |
| Instance | GET http://localhost:3000/rest/products/search?q= |
| Asset | Application |
| Countermeasure | Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side.If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.If database Stored Procedures can be used, use them.Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!Do not create dynamic SQL queries using simple string concatenation.Escape all data received from the client.Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.Apply the privilege of least privilege by using the least privileged database user possible.In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.Grant the minimum database access that is necessary for the application. |
| Attack | Command Line Execution through SQL Injection; Object Relational Mapping Injection; SQL Injection through SOAP Parameter Tampering; Expanding Control over the Operating System from the Database; SQL Injection; Blind SQL Injection |
| Likelihood Of Exploit | High |
| Scan tool | ZAP |
| Test result | FAIL |
| Timestamp | 29/08/2020 09:28 UTC |
| ID | 71dde7ab1440a61ef30b3f844d43b929 |
| NIST Security Control | SI-10 |

| Low | Private IP Disclosure |
|---|---|
| Weakness | CWE-200: Exposure of Sensitive Information to an Unauthorized Actor. The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information. |
| Threat | Read Application Data |
| Instance | GET http://localhost:3000/rest/admin/application-configuration |
| Asset | Application |
| Countermeasure | Remove the private IP address from the HTTP response body. For comments, use JSP/ASP/PHP comment instead of HTML/JavaScript comment which can be seen by client browsers. |
| | Excavation; Subverting Environment Variable Values; Footprinting; Exploiting Trust in Client; |

Figura 5.3: Estratto report Juice Shop v.11.1.3

| Risk | Number of occurrences |
|---|---|
| medium | 104 |
| critical | 6 |
| high | 7 |
| low | 37 |
| Total: | 154 |

| Failing NIST Security Controls |
|---|
| SI-11, AC-3, SI-10, SC-8, SC-13 |

Figura 5.4: Statistiche Juice Shop v.11.1.3

### 5.4.1 Applicazione delle mitigazioni proposte v.11.1.3-1

Vengono manualmente applicate alcune delle mitigazioni proposte e viene eseguito un nuovo push. Allo stesso tempo, viene effettuato il building del sistema generando una nuova versione dell'immagine Docker.

In particolare, così come proposto dal report generato, per quanto riguarda la weakness di "SQL Injection", rilevata sull'endpoint '*GET http://localhost:3000/rest/products/search*', l'input dell'utente deve essere sottoposto ad un processo di validazione prima di poter essere utilizzato come parametro della query (Figura 5.5).

Si riporta un estratto del codice originale di *router/search.js*:

```
let criteria = req.query.q [..]
```

```
models.sequelize.query(`SELECT * FROM Products WHERE
   ((name LIKE '%${criteria}%' OR description LIKE
   '%${criteria}%') AND deletedAt IS NULL) ORDER BY name`)
```

Per la validazione della stringa si può far affidamento ad una libreria terza, come ad esempio *sqlstring* [32], che si occupa di effettuare l'escape di stringhe da utilizzare come parametro per query SQL:

```
let criteria = req.query.q [..]
criteria = SqlString.escape(criteria)
models.sequelize.query(`SELECT * FROM Products WHERE
   ((name LIKE '%${criteria}%' OR description LIKE
   '%${criteria}%') AND deletedAt IS NULL) ORDER BY name`)
```

Per quanto riguarda "Hidden File Found" si rimuovono i file non necessari in produzione e si rendono nascosti quelli, invece, necessari (Figura 5.6).

| High | SQL Injection - SQLite |
|---|---|
| Weakness | CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. |
| Threat | Read Application Data; Bypass Protection Mechanism; Modify Application Data |
| Instance | GET http://localhost:3000/rest/products/search?q= |
| Asset | Application |
| Countermeasure | Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side.If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.If database Stored Procedures can be used, use them.Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!Do not create dynamic SQL queries using simple string concatenation.Escape all data received from the client.Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.Apply the privilege of least privilege by using the least privileged database user possible.In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.Grant the minimum database access that is necessary for the application. |

Figura 5.5: Contromisura proposta SQL Injection

| Medium | Hidden File Found |
|---|---|
| Weakness | CWE-538: Insertion of Sensitive Information into Externally-Accessible File or Directory. The product places sensitive information into files or directories that are accessible to actors who are allowed to have access to the files, but not to the sensitive information. |
| Threat | Read Files or Directories |
| Instance | GET http://localhost:3000/.svn/entries |
| Asset | Application |
| Countermeasure | Consider whether or not the component is actually required in production, if it isn't then disable it. If it is then ensure access to it requires appropriate authentication and authorization, or limit exposure to internal systems or specific source IPs, etc. |

Figura 5.6: Contromisura proposta Hidden File Found

Lo strumento è avviato con il seguente comando:

```
sh run.sh "Juice Shop" 11.1.3-1 fixCommitId myjuice
   11.1.3-1 http:\\localhost:3000 anchore zap
```

Di seguito si riportano le statistiche di questa seconda analisi assieme ad un estratto del report di sicurezza ottenuto.

**Juice Shop v. 11.1.3-1 (fixCommitId) Security Report**

| Medium | CVE-2019-15847 |
|---|---|
| Weakness | CWE-331: Insufficient Entropy. The software uses an algorithm or scheme that produces insufficient entropy, leaving patterns or clusters of values that are more likely to occur than others. |
| Threat | Bypass Protection Mechanism |
| Instance | libgcc-9.2.0-r4 |
| Asset | Container |
| Countermeasure | Phase: Implementation Determine the necessary entropy to adequately provide for randomness and predictability. This can be achieved by increasing the number of bits of objects such as keys and seeds. |
| Attack | Session Credential Falsification through Prediction |
| Likelihood Of Exploit | Unknown |
| Scan tool | Anchore |
| Test result | UNKNOWN |
| Timestamp | 29/08/2020 10:06 UTC |
| ID | fd12571bf5d173bf07f3bcbf5764c5f9 |
| NIST Security Control | SC-13 |

| High | SQL Injection - SQLite |
|---|---|
| Weakness | CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. |
| Threat | Read Application Data; Bypass Protection Mechanism; Modify Application Data |
| Instance | GET http://localhost:3000/rest/products/search?q= |
| Asset | Application |
| Countermeasure | Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side.If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.If database Stored Procedures can be used, use them.Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!Do not create dynamic SQL queries using simple string concatenation.Escape all data received from the client.Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.Apply the privilege of least privilege by using the least privileged database user possible.In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.Grant the minimum database access that is necessary for the application. |
| Attack | Command Line Execution through SQL Injection; Object Relational Mapping Injection; SQL Injection through SOAP Parameter Tampering; Expanding Control over the Operating System from the Database; SQL Injection; Blind SQL Injection |
| Likelihood Of Exploit | High |
| Scan tool | ZAP |
| Test result | PASS |
| Date | 18/08/2020 22:35 UTC |
| ID | 71dde7ab1440a61ef30b3f844d43b929 |
| NIST Security Control | SI-10 |

| Medium | Hidden File Found |
|---|---|
| Weakness | CWE-538: Insertion of Sensitive Information into Externally-Accessible File or Directory. The product places sensitive information into files or directories that are accessible to actors who are allowed to have access to the files, but not to the sensitive information. |
| Threat | Read Files or Directories |
| Instance | GET http://localhost:3000/.svn/entries |
| Asset | Application |
| Countermeasure | Consider whether or not the component is actually required in production, if it isn't then disable it. If it is then ensure access to it requires appropriate authentication and authorization, or limit exposure to internal systems or specific source IPs, etc. |
| Attack | WSDL Scanning |

Figura 5.7: Estratto report Juice Shop v.11.1.3-1

| Risk | Number of occurrences |
|---|---|
| medium | 69 |
| high | 6 |
| critical | 6 |
| low | 36 |
| **Total:** | 117 |

| Failing NIST Security Controls |
|---|
| SC-13, SI-10, SC-8, AC-3 |

Figura 5.8: Statistiche Juice Shop v.11.1.3-1

Si può notare come, questa volta, i test per "*SQL Injection*" e "*Hidden File Found*" terminano con successo, per cui le vulnerabilità risulterebbero essere mitigate.

### 5.4.2 Applicazione ulteriori modifiche al sistema v.11.1.3-2

Vengono introdotte nuove funzionalità al sistema e viene eseguito un nuovo commit e push. Lo strumento è avviato con il seguente comando:

```
sh run.sh "Juice Shop" 11.1.3-2 myCommitId myjuice
   11.1.3-2 http:\\localhost:3000 anchore zap
```

Di seguito si riportano le statistiche di questa ulteriore analisi.

| Risk | Number of occurrences |
|---|---|
| medium | 69 |
| high | 6 |
| critical | 6 |
| low | 36 |
| **Total:** | 117 |

| Failing NIST Security Controls |
|---|
| SC-13, SI-10, SC-8, AC-3 |

Figura 5.9: Statistiche Juice Shop v.11.1.3-2

In questo caso, le modifiche non hanno impattato sulla sicurezza del sistema, per cui il report non è variato, a meno di non mostrare più le rilevazioni per i problemi di sicurezza risolti al paragrafo precedente.

### 5.4.3 Applicazione modifiche al sistema, con regressione v.11.1.3-3

Vengono introdotte nuove funzionalità al sistema e viene eseguito un nuovo commit e push.

Le modifiche hanno volutamente reintrodotto problemi di SQL Injection su di un endpoint fixato in precedenza.

```
sh run.sh "Juice Shop" 11.1.3-3 regCommitId myjuice
   11.1.3-3 http:\\localhost:3000 anchore zap
```

Di seguito si riportano le statistiche di questa ultima analisi assieme ad un estratto del report di sicurezza ottenuto.

| **High** | **SQL Injection - SQLite** |
|---|---|
| Weakness | CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. |
| Threat | Read Application Data; Bypass Protection Mechanism; Modify Application Data |
| Instance | GET http://localhost:3000/rest/products/search?q= |
| Asset | Application |
| Countermeasure | Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side.If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'If the application uses ASP, use ADO Command Objects with strong type checking and parameterized queries.If database Stored Procedures can be used, use them.Do *not* concatenate strings into queries in the stored procedure, or use 'exec', 'exec immediate', or equivalent functionality!Do not create dynamic SQL queries using simple string concatenation.Escape all data received from the client.Apply a 'whitelist' of allowed characters, or a 'blacklist' of disallowed characters in user input.Apply the privilege of least privilege by using the least privileged database user possible.In particular, avoid using the 'sa' or 'db-owner' database users. This does not eliminate SQL injection, but minimizes its impact.Grant the minimum database access that is necessary for the application. |
| Attack | Command Line Execution through SQL Injection; Object Relational Mapping Injection; SQL Injection through SOAP Parameter Tampering; Expanding Control over the Operating System from the Database; SQL Injection; Blind SQL Injection |
| Likelihood Of Exploit | High |
| Scan tool | ZAP |
| Test result | FAIL |
| Timestamp | 29/08/2020 11:17 UTC |
| ID | 71dde7ab1440a61ef30b3f844d43b929 |
| **Regression** | **11.1.3-1 (fixCommitId)** |
| NIST Security Control | SI-10 |

Figura 5.10: Estratto report Juice Shop v.11.1.3-3

| Risk | Number of occurrences |
|---|---|
| medium | 69 |
| high | 7 |
| critical | 6 |
| low | 36 |
| **Total:** | 118 |

| Failing NIST Security Controls |
|---|
| SI-10, AC-3, SC-13, SC-8 |

Figura 5.11: Statistiche Juice Shop v.11.1.3-3

Si può notare come lo strumento non solo rileva nuovamente il problema di sicurezza ma si accorge anche della regressione, segnalando la versione dell'applicazione (v.11.1.3-1), ed il riferimento al commit (fixCommitId), che ha, in precedenza, risolto il problema.

## 5.5 Analisi andamento storico generale

A valle di ciascuna esecuzione dello strumento, si genera una timeline storica che mostra l'andamento del numero di rilevazioni nel tempo, in funzione della versione del sistema (e relativo commit).

Di seguito si riporta uno screenshot del report storico generato.

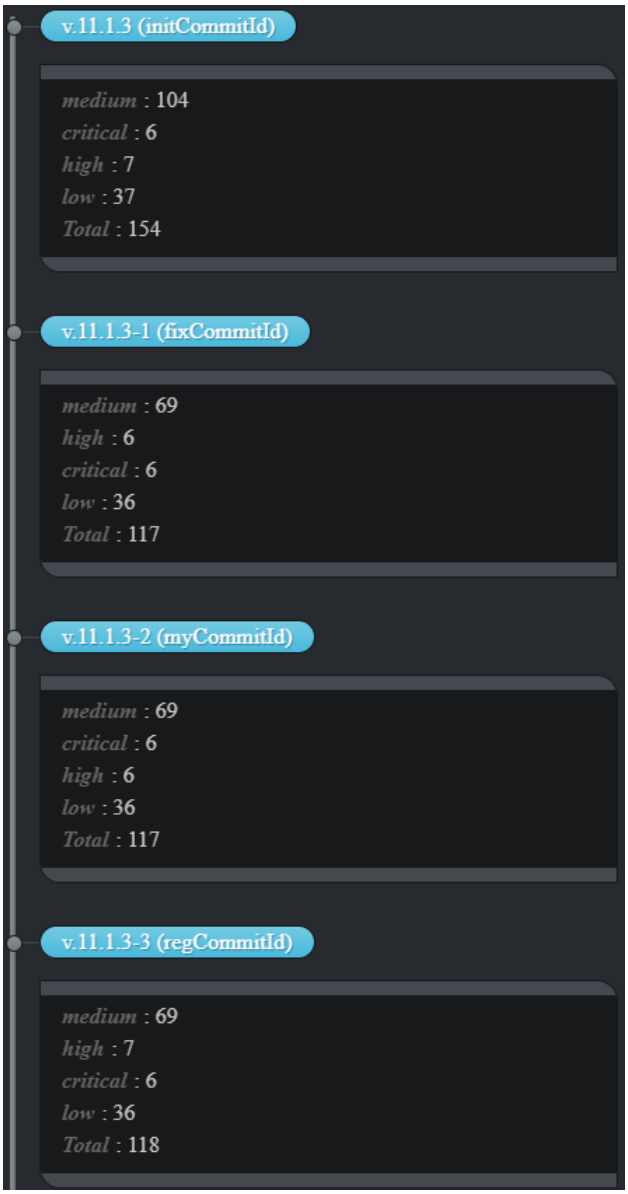


Figura 5.12: Timeline storico Juice Shop

## 5.6 Analisi andamento storico rilevazioni

A titolo di esempio si riporta l'andamento storico della rilevazione relativa a "*SQL Injection*", rilevata nella prima versione dell'applicazione (v.11.1.3), risolta con la seconda versione (v.11.1.3-1) e reintrodotta

con l'ultima (v.11.1.3-3).

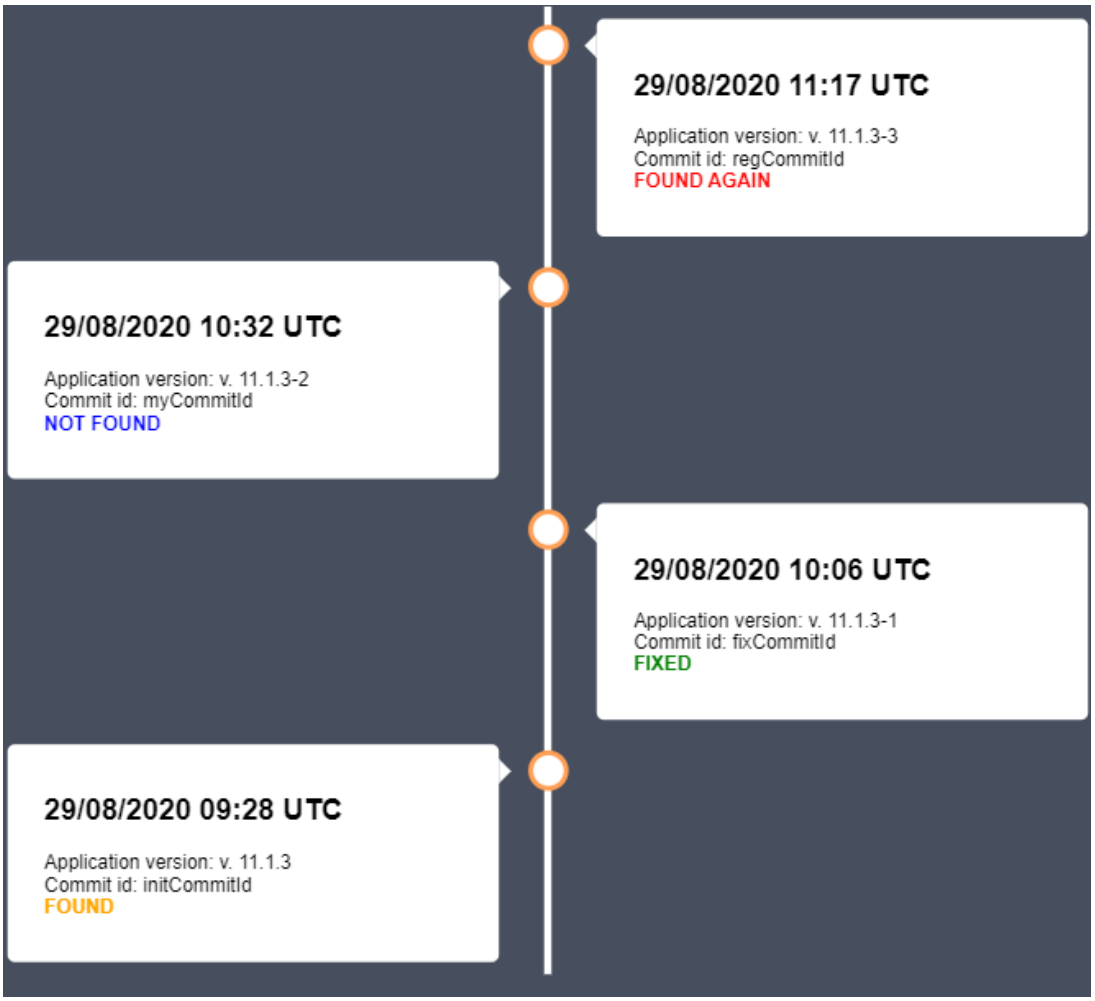


Figura 5.13: Timeline SQL Injection Juice Shop

## 5.7 Discussione risultati

Lo strumento è stato in grado di rilevare alcune delle vulnerabilità del sistema e proporre delle contromisure da adottare. Come mostrato al Paragrafo 5.4.1 applicando le contromisure proposte è stato possibile mitigare i problemi di "SQL Injection" e "Hidden File Found".

Per ciascuna rilevazione è possibile visualizzare l'andamento al variare delle versioni dell'applicazione, e rilevare problemi di regressione. Mantenendo anche un riferimento ai commit corrispondenti, lo sviluppatore può facilmente tenere traccia delle modifiche che portano alla risoluzione, o riesposizione, di una determinata vulnerabilità. Come mostrato al Paragrafo 5.4.3 lo strumento è stato in grado di rilevare la regressione della "SQL Injection" sull'endpoint */search* della web app.

Lo storico generale del sistema fornisce, infine, delle metriche che possono essere prese in considerazione per un eventuale security assessment del sistema. Al Paragrafo 5.6 è stato evidenziato come la versione 11.1.3-1 dell'applicazione presenti meno problemi delle altre. Tuttavia, i controlli di sicurezza definiti dal NIST 800-53 non soddisfatti restano comunque gli stessi.

# Capitolo 6

# Caso di studio: Sito web basato su Joomla!

Con l'intento di mostrare la flessibilità della metodologia proposta, oltre che dello strumento realizzato, si riporta un altro esempio di utilizzo.

L'esempio in questione cerca di mostrare come lo strumento possa mostrarsi utile anche per la valutazione della sicurezza di sistemi il cui sviluppo è da intendersi come integrazione e configurazione di componenti già sviluppati. Lo strumento non solo è in grado di analizzare il sistema e riportare l'elenco delle vulnerabilità riscontrate, con i relativi controlli di sicurezza NIST 800-53 falliti, ma con l'ausilio di strumenti terzi di penetration testing riesce anche a testare l'effettiva exploitabilità delle weakness riscontrate. Anche in questo caso, la metodologia (e dunque lo strumento) riesce a mostrarsi utile, oltre che di semplice

utilizzo e integrazione.

In questa casistica, chiaramente, le contromisure proposte a livello del codice potrebbero non essere applicate, trattandosi di uno sviluppo per mera integrazione di componenti. Tuttavia, il report riporta, ove possibile, i componenti (istanze) vulnerabili per cui si può pensare di aggiornarli con la speranza che gli sviluppatori abbiano mitigato la vulnerabilità nota.

Una ulteriore differenza consiste nel riferimento al commit che, in questo caso, non è determinato. Pertanto, le evoluzioni sono caratterizzate in funzione della sola versione associata al sistema.


Figura 6.1: Joomla! Content Management System

Il sistema software analizzato è un sito web containerizzato, realizzato con Joomla! [34], un noto content management system (CMS) per la realizzazione di siti web, che fa uso di un database MySQL e di un server SMTP per l'invio di email agli utenti.

In questo caso di tratta di un sito web Joomla! containerizzato, per cui vengono selezionati *Anchore* e *JoomScan* come strumenti per la fase di scansione del sistema.

## 6.1 Configurazione ambiente

Per poter riprodurre l'esperimento occorre scaricare e installare i seguenti software:

- Docker 19.03.12
- Python 3.6.9
- JoomScan 0.0.7
- Immagine Docker anchore/inline-scan:v0.8.1
- Immagine Docker joomla:3.4.3
- Immagine Docker mysql:5.7.31
- Immagine Docker reachfive/fake-smtp-server:0.8.1
- Package Python requests-html 0.10.0

## 6.2 Descrizione del sistema mediante il formalismo MACM

Il sito web è in esecuzione all'interno di un container, in uno stack Docker installato su di una macchina virtuale gestita da una piattaforma di virtualizzazione (Proxmox) ospitata in un'infrastruttura cloud, presso l'Università della Campania "Luigi Vanvitelli" (Aversa, Caserta, Italia)

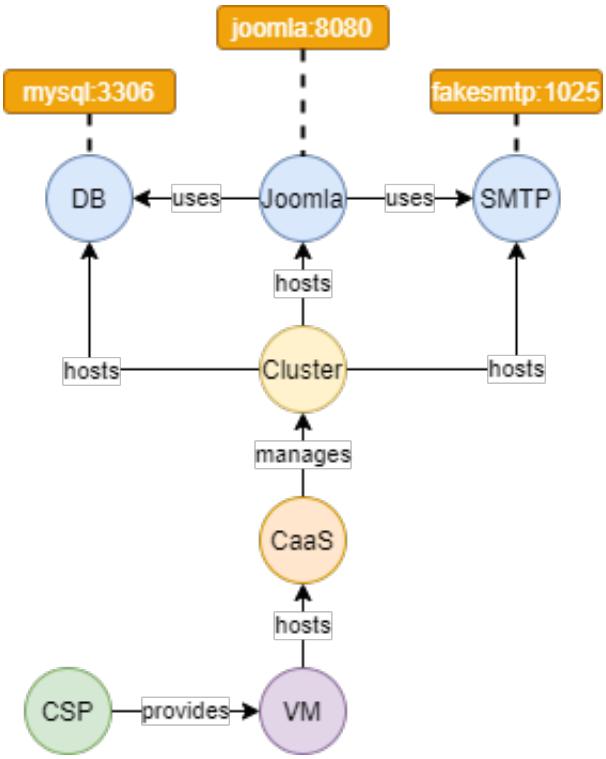


Figura 6.2: MACM Sito web Joomla!

## 6.3 Configurazione dello strumento

La configurazione prevede principalmente la scelta degli strumenti di scansione da utilizzare, tra quelli integrati. In questo caso di tratta di un sito web Joomla! containerizzato, per cui si richiedono *Anchore* e *JoomScan*.

Inoltre, ad ogni esecuzione (manuale o automatizzata) occorre passare in input la versione dell'applicazione e "noCommit", come identificativo del commit di riferimento, poiché non facciamo uso di un sistema di versionamento.

Il comando sarà dunque:

```
sh run.sh "My website" [versione applicazione] noCommit
   mywebsite [versione immagine] http:\\localhost:8080
   anchore joomscan
```

## 6.4 Analisi cicli di evoluzione del sistema

Una prima esecuzione dello strumento è effettuata in seguito alla configurazione iniziale del sito, a valle di una nuova installazione di Joomla! v. 3.4.3 .

```
sh run.sh "My website" 1.0 noCommit mywebsite 1.0
   http:\\localhost:8080 anchore joomscan
```

Di seguito si riportano le statistiche di questa prima analisi assieme ad un estratto del report di sicurezza ottenuto.

| Risk | Number of occurrences |
|---|---|
| negligible | 74 |
| medium | 112 |
| high | 41 |
| low | 1 |
| unknown | 2 |
| **Total:** | 230 |

| Failing NIST Security Controls |
|---|
| AC-3, SI-10, SC-4, SC-23, SC-8 |

Figura 6.3: Statistiche prima analisi sito Joomla!

| High | CVE-2019-5482 |
|---|---|
| Weakness | CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow. |
| Threat | Modify Memory; Execute Unauthorized Code or Commands; DoS: Crash, Exit, or Restart; DoS: Resource Consumption (CPU) |
| Instance | curl-7.38.0-4+deb8u2 |
| Asset | Application |
| Countermeasure | Phase: Requirements Strategy: Language Selection Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer. Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe. |
| Attack | Buffer Overflow via Environment Variables; Overflow Buffers; Client-side Injection-induced Buffer Overflow; Filter Failure through Buffer Overflow; MIME Conversion; Overflow Binary Resource File; Buffer Overflow via Symbolic Links; Overflow Variables and Tags; Buffer Overflow via Parameter Expansion; String Format Overflow in syslog(); Buffer Overflow in an API Call; Buffer Overflow in Local Command-Line Utilities; Forced Integer Overflow |
| Likelihood Of Exploit | High |
| Scan tool | Anchore |
| Test result | UNKNOWN |
| Timestamp | 02/09/2020 23:36 UTC |
| ID | 6d32546fd3b73cce7e3558445b5e7eb0 |
| NIST Security Control | SI-10 |

| High | PHPMailer Remote Code Execution Vulnerability (CVE-2016-10033) |
|---|---|
| Weakness | CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection'). The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component. |
| Threat | Execute Unauthorized Code or Commands |
| Instance | http://localhost:8080/ |
| Asset | Application |
| Countermeasure | Phase: Architecture and Design If at all possible, use library calls rather than external processes to recreate the desired functionality. |
| Attack | PHPMailer < 5.2.20 - Remote Code Execution |
| Likelihood Of Exploit | High |
| Scan tool | joomscan |

Figura 6.4: Report prima analisi sito Joomla!

In Figura 6.4 si può notare come le vulnerabilità siano a livello di alcuni componenti utilizzati, come ad esempio *PHPMailer* in versione inferiore alla 5.2.20, vulnerabile alla *Remote Code Execution*. Analogamente, da *Anchore*, viene rilevata una vulnerabilità sul pacchetto *curl-7.38.0-4+deb8u2*. Non volendo (o potendo) accedere al codice di questi componenti, per risolvere il problema, si può pensare di aggiornarli ad una versione successiva.

Si procede dunque con l'aggiornamento di alcuni dei componenti vulnerabili.

### 6.4.1 Analisi nuova versione del sistema

Terminato l'aggiornamento di alcuni dei componenti rilevati come vulnerabili, si avvia nuovamente lo strumento.

```
sh run.sh "My website" 1.1 noCommit mywebsite 1.1
   http:\\localhost:8080 anchore joomscan
```

Di seguito si riportano le statistiche di questa seconda analisi assieme alla timeline storica.

| Risk | Number of occurrences |
|---|---|
| negligible | 74 |
| medium | 111 |
| high | 37 |
| low | 1 |
| **Total:** | 223 |

| Failing NIST Security Controls |
|---|
| SI-10, SC-23, SC-4, AC-3, SC-8 |

Figura 6.5: Statistiche seconda analisi sito Joomla!

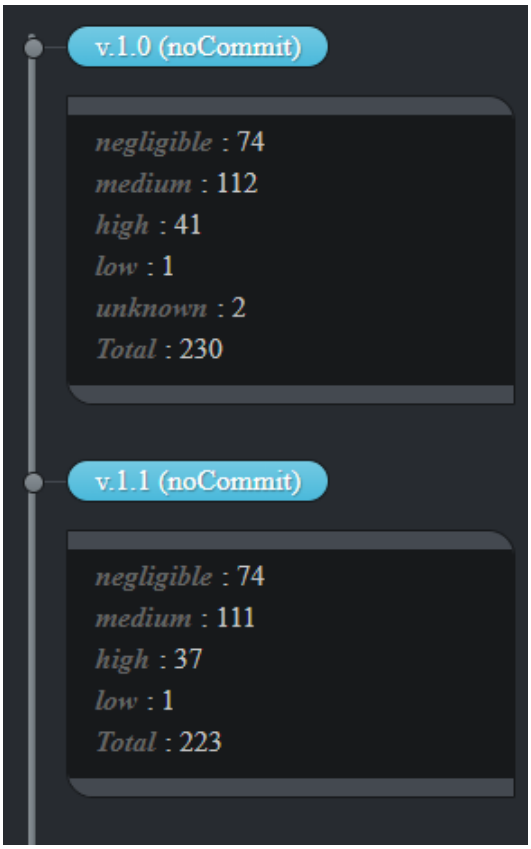


Figura 6.6: Timeline storica sito Joomla!

## 6.5 Discussione risultati

Lo strumento è stato in grado di rilevare alcune delle vulnerabilità del sistema e proporre delle contromisure da adottare a livello dell'implementazione e ad evidenziare i componenti vulnerabili. Come descritto al Paragrafo 6.4.1 aggiornando alcuni dei componenti rilevati come vulnerabili è stato possibile mitigare i problemi segnalati.

Lo storico generale del sistema fornisce delle metriche che possono essere prese in considerazione per un security assessment del sistema. La Figura 6.6 evidenzia come la seconda versione del sistema presenti meno problemi, rispetto alla precedente. Tuttavia, i controlli di sicurezza definiti dal NIST 800-53 non soddisfatti restano comunque gli stessi (Figure 6.3 e 6.5).

# Capitolo 7

# Conclusioni

Il penetration testing si propone tra le principali soluzioni al testing di sicurezza ed è, attualmente, ampiamente utilizzato da team specializzati per controllare la sicurezza delle applicazioni. Allo stato attuale, tuttavia, non è ben integrato nei moderni processi di sviluppo software, poiché la sua introduzione, ad ogni iterazione, risulterebbe eccessivamente complessa oltre che costosa, vista la necessità di personale altamente qualificato. Per colmare questo gap, è stata proposta una metodologia di testing semi-automatico, semplice da utilizzare, con l'obiettivo di ottenere una valutazione a grana grossa della sicurezza del sistema, fin dai primi prototipi, a basso costo.

Il risultato di maggior rilievo è che, grazie all'automazione introdotta, anche un team di sviluppo senza security skills è in grado di valutare e monitorare la sicurezza di un sistema in via di sviluppo, in poco tempo.

L'integrazione della metodologia con i moderni processi di sviluppo software è semplice, lo strumento non necessita di configurazioni poiché sfrutta strumenti terzi sia per la fase di scansione, che prende in input l'applicazione deployata, sia per la fase di testing in cui i testing tools sono orchestrati automaticamente dallo strumento che implementa la metodologia, come quello proposto al Capitolo 4.

Il mapping tra le weakness ed i controlli del NIST 800-53, riportato nel report finale dello strumento proposto al Capitolo 4 in termini di controlli che non sono rispettati, si mostra di notevole aiuto in contesti di security assessment del sistema: in maniera semplice ed automatica è possibile tenere traccia di quali controlli vengono meno e quali sono le weakness che causano questo effetto. Allo stesso modo, il mapping fa sì che si riesca a risalire alla weakness che causa il fallimento di un controllo e la contromisura proposta suggerisce una o più strategie per mitigarla. In aggiunta, lo storico generale produce delle ulteriori metriche che possono essere prese in considerazione in contesti valutativi.

In conclusione, si sottolinea che automatizzare il processo di valutazione della sicurezza è tanto importante quanto utile ma resta comunque un problema a cui è difficile dare una soluzione esaustiva. Il processo può basarsi su numerosi standard, controlli di sicurezza, database di vulnerabilità note, attacchi noti etc., tuttavia il numero di minacce e vulnerabilità imprevedibili è molto elevato per cui sono

sempre richieste tecniche estremamente flessibili di analisi del rischio.

# Capitolo 8

# Sviluppi futuri

La metodologia è priva di una procedura di generazione automatica degli attacchi da sferrare nella fase di testing e, sebbene essa sia indipendente dalla tipologia di applicazione che si vuole analizzare, la scelta degli strumenti può influenzare il risultato della analisi e dei test. Gli strumenti terzi da integrare, per le fase di scansione e testing, possono essere più o meno specifici rispetto alla natura dell'applicazione. Il caso di studio riportato, ad esempio, ha previsto l'integrazione di OWASP ZAP, uno strumento specifico per le applicazioni web, che risulterebbe pressoché inutile per applicazioni di altra tipologia. Discorso analogo per JoomScan, specifico per siti web Joomla!-based. Una proposta di evolutiva potrebbe essere quella di progettare una procedura di generazione automatica degli attacchi abbandonando di fatto la dipendenza dagli strumenti esterni.

Sempre per quanto riguarda la fase di testing, basata o meno su

strumenti terzi, si potrebbe progettare una logica più robusta di generazione dinamica degli attacchi, dipendente dall'architettura del sistema. Gli stessi test potrebbero essere prioritizzati assegnando, ad esempio, dei pesi alle vulnerabilità e/o sfruttando lo storico delle rilevare per assegnare la priorità ai test successivi.

Nell'ottica di integrazione della metodologia, e dello strumento, in un processo di security assessment, si potrebbe prevedere l'aggiunta di una fase iniziale in cui si analizzano i requisiti di sicurezza del sistema. Si potrebbe, ad esempio, utilizzare un sondaggio per richiedere quali controlli di sicurezza (NIST 800-53) si vuole soddisfare. Così facendo, lo strumento potrebbe stabilire, in maniera automatica, se esistono motivazioni note tali per cui uno o più controlli richiesti non risulterebbero rispettati. Questa stessa strategia potrebbe essere migliorata ampliando il modello di mapping CWE-NIST con i "*Control Enhancements*". Ciò consentirebbe allo strumento, in aggiunta alla verifica dei controlli, di classificare il sistema in uno dei 3 livelli del NIST: LOW, MED e HIGH.

Il modello di sicurezza potrebbe, inoltre, essere esteso al fine di supportare multipli standard di sicurezza e non solo il NIST Control Framework SP-800-53, ad esempio per semplificare e rendere più o meno automatici processi di certificazione.

Al fine di fornire un livello di rischio più appropriato al contesto cui si colloca l'applicazione in esame, si potrebbe prevedere un maggior

grado di personalizzazione dello strumento offrendo la possibilità di integrare delle metriche custom per la definizione del livello di rischio delle rilevazioni, in aggiunta a quella fornita da MITRE CWE.

# Bibliografia


[1] P. Mishra A. Tomar, D. Jeena and R. Bisht. *Docker Security: A Threat Model, Attack Taxonomy and Real-Time Attack Scenario of DoS*. 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2020, pp. 150-155, 2020.

[2] Anchore. *Anchore Container Analysis*. https://anchore.com/opensource/, Agosto 2020.

[3] P. Bogaerts. *Arp Spoofing Docker Containers*. https://dockersec.blogspot.com/2017/01/arp-spoofing-dockercontainers_26.html, 2017.

[4] Perini A. Giorgini P. et al. Bresciani, P. *Tropos: An Agent-Oriented Software Development Methodology*. Autonomous Agents and Multi-Agent Systems 8, 203–236, 2004.

[5] T. Bui. *Analysis of Docker Security*. Aalto University T-110.5291 Seminar on Network Security, 2014.

[6] De Benedictis A. Rak M. Casola, V. and U. Villano. *A methodology for automated penetration testing of cloud applications.* Int. J. Grid and Utility Computing, Vol. 11, No. 2, pp.267–277, 2020.

[7] De Benedictis A. Rak M. Casola, V. and U. Villano. *A novel Security-by-Design methodology: Modeling and assessing security by SLAs with a quantitative approach.* The Journal of Systems and Software, 2020.

[8] De Benedictis A. Rak M. Rios E. Casola, V. *Security-by-design in clouds: a security-sla driven methodology to build secure cloud applications.* Procedia Computer Science, Proceedings of the 2nd International Conference on Cloud Forward: From Distributed to Complete Computing, Vol. 97, pp.53–62., 2016.

[9] Chanliau M.D. Cavoukian, A. *Privacy and Security by Design: An Enterprise Architecture Approach.* Information and Privacy Commissioner, 2013.

[10] National Cyber Security Centre. *Secure by Default.* https://www.ncsc.gov.uk/information/secure-default, 2018.

[11] B. Kimminich ed altri volontari. *OWASP Juice Shop.* https://owasp.org/www-project-juice-shop/, 2020.

[12] Ferreira A. Rajamony R. Rubio J. Felter, W. *An updated performance comparison of virtual machines and linux containers.* Technology 28, 32 (2014), 2014.

[13] Pina J. Borges G. Martins J. Dias N. Gomes H. Manuel C. Gomes, J. *Exploring Containers for Scientific Computing.* XSEDE16: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, 2016.

[14] Red Hat. *Cos'è SELinux?* https://www.redhat.com/it/topics/linux/what-is-selinux, 2020.

[15] I Yasin I Ghani. *Software Security Engineering in Extreme Programming Methodology: A Systematic Literature Review.* Science International, 2013.

[16] S. A. Kumar J. Chelladhurai, P. R. Chelliah. *Securing Docker containers from DOS attacks.* IEEE Int. Conf. Services Comput., pp. 856-859, 2016.

[17] N. Kumar K. Kaur, T. Dhand and S. Zeadally. *Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers.* IEEE Wireless Commun., vol. 24, no. 3, pp. 48-56, 2017.

[18] D. Merkel. *Docker: lightweight Linux containers for consistent development and deployment.* Linux J. 2014(239), 2 (2014), 2014.

[19] MITRE. *MITRE CWE-NIST Mapping.* https://github.com/mitre/heimdall_tools/blob/master/lib/data/cwe-nist-mapping.csv, Agosto 2020.

[20] V. Mohan and L. B. Othmane. *SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps.* 11th International Conference on Availability, Reliability and Security (ARES), Salzburg, pp. 542-547, 2016.

[21] Sani N.F.M. Almasi M.M. Mougouei, D. *S-Scrum : a Secure Methodology for Agile Development of Web Services.* World of Computer Science and Information Technology Journal (WSCIT) 3 (1), 15–19, 2013.

[22] NIST. *NIST Special Publication 800-53.* https://nvd.nist.gov/800-53, Agosto 2020.

[23] Nyantec. *Docker networking considered harmful.* https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful, 2015.

[24] Rahman M. M. ben Othmane L. Ghani I. & Arbain A. F Oueslati, H. *Evaluation of the Challenges of Developing Secure Software Using the Agile Approach.* International Journal of Secure Software Engineering (IJSSE), 7(1), 17-37, 2016.

[25] OWASP. *OWASP Zed Attack Proxy.* https://www.zaproxy.org/, Agosto 2020.

[26] Christoph Pohl and Hans-Joachim Hof. *Secure Scrum: Development of Secure Software with Scrum.* arXiv:1507.02992, 2015.

[27] Christoph Pohl and Hans-Joachim Hof. *Secure Scrum: Development of Secure Software with Scrum.* arXiv:1507.02992, 2015.

[28] rezasp. *JoomScan.* https://github.com/rezasp/joomscan, 2018.

[29] J. Rudenberg. *Docker Image Insecurity.* https://titanous.com/posts/docker-insecurity, 2014.

[30] T. Dimitriou S. Sultan, I. Ahmad. *Container Security: Issues Challengesand the Road Ahead.* IEEE Access, vol. 7, pp. 52976-52996, 2019.

[31] P. Felber M. Pasin V. Schiavoni S. Vaucher, R. Pires and C. Fetzer. *SGX-Aware container orchestration for heterogeneous clusters.* Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS), pp. 730-741, 2018.

[32] sqlstring. *sqlstring.* https://github.com/mysqljs/sqlstring, 2020.

[33] R. Di Pietro T. Combe, A. Martin. *To docker or not to docker: A security perspective.* IEEE Cloud Comput., vol. 3, no. 5, pp. 54-62, 2016.

[34] OSM Development Team. *Joomla! Content Management System (CMS).* https://www.joomla.org/, 2005.

[35] J. Turnbull. *The Docker book.* James Turnbull; 18092 edizione, 2014.

[36] Williams L. Ur Rahman, A.A. *Software security in DevOps: synthesizing practitioners' perceptions and practices.* CSED '16: Proceedings of the International Workshop on Continuous Software Evolution and, 2016.

[37] D. Walsh. *Are Docker Containers Really Secure?* Dec. 27, 2017; https://opensource.com/business/14/7/docker-security-selinux, 2017.

[38] Bodén M. Boström G. Wäyrynen, J. *Security Engineering and eXtreme Programming: An Impossible Marriage? In: Extreme Programming and Agile Methods.* XP/Agile Universe, pp. 117–128, 2004.

[39] R. Yasrab. *Mitigating Docker Security Issues.* arXiv:1804.05039, 2018.

[40] I. Ghani Z. Azham and N. Ithnin. *Security backlog in Scrum security practices.* Malaysian Conference in Software Engineering, Johor Bahru, pp. 414-41, 2011.