Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**

**Corso di Laurea in Informatica**

# RICERCA DI VULNERABILITÀ IN APPLICAZIONI APPLE SU DISPOSITIVI MOBILE

**Relatore:**
**Prof. Marco Prandini**

**Presentata da:**
**ADRIANO PACE**

**Sessione II**
**Anno Accademico 2021/2022**

*A mia cugina che ha sempre vegliato su di me, ovunque io sia stato, ovunque lei sia. . . .*

# Introduzione

Nell'anno 1983, esattamente 39 anni fa, veniva commercializzato il primo smartphone, chiamato **DynaTAC 8000x**, esso non aveva connessione internet, non aveva touch-screen, pesava circa 1kg e non permetteva alcun tipo di riproduzione multimediale.
Da quel momento in poi, l'evoluzione dei telefoni cellulari ha subito una rapida impennata, fino ad arrivare ai nostri giorni, in cui un dispositivo mobile chiamato smartphone rappresenta un elemento fondamentale della nostra quotidianità, praticamente essenziale. Sul nostro smartphone siamo soliti avere tutto, dalle nostre foto private ai nostri dati bancari e le nostre password. Quello che però molti di noi non sono soliti avere è la giusta attenzione all'aspetto di sicurezza che circonda tutte queste cose.
Sappiamo quanto veramente vulnerabile sia il software presente sul nostro dispositivo ? Possiamo fidarci ciecamente delle applicazioni che installiamo ? Chi ci dice che l'applicazione della nostra banca sia veramente sicura ?
La sicurezza spesso trascurata di questi dispositivi ci espone continuamente al rischio di cyberattacchi, o (negli ultimi tempi sempre più in aumento [12]) a ricatti di ingegneria sociale.
Un dispositivo mobile può risultare vulnerabile sin dalla sua nascita, più di una volta è diventato di pubblico dominio il caso in cui aziende produttrici di dispotivi mobile venissero accusate di compiacere le regole di governi autoritari inserendo programmi chiamati

**spyware** (software spia) per aumentare il controllo sulla popolazione [16].
Si rifletta anche sul numero sempre più crescente (secondo le statistiche) del numero di cellulari dimenticati in luoghi pubblici. Molti utenti, per pigrizia o ignoranza, non utilizzano un sistema di autenticazione base sul proprio dispositivo (scansione biometrica dell'impronta o puzzle piuttosto che pin), rendendosi totalmente vulnerabili.
L'educazione alla cultura della sicurezza informatica sembra essere ancora oggi sfortunatamente un qualcosa di nicchia, riservata agli esperti del settore.
Il progetto raccontato in questo documento affronta la ricerca di vulnerabilità di applicazioni software in ambiente mobile destinate al mercato Apple, con un occhio di riguardo ad applicazioni di private internet banking, ritenute più sensibili nel contesto di sicurezza informatica.

# Indice

# Elenco delle figure

# Capitolo 1

# Sicurezza delle applicazioni software mobile Apple e Android

È parer comune, nel mondo della sicurezza informatica e non solo, credere che i sistemi operativi Apple godano di una maggiore sicurezza rispetto a quelli Android, nello specifico che siano meno permeabili ed attaccabili dalle minacce del mondo esterno. Ma è davvero così ?

## 1.1 Open source software vs Closed source software

Si parla di software **Open source** quando il codice sorgente di applicazione è consultabile gratuitamente da tutti gli utenti di quest'ultima e non solo. In un progetto Open source, chiunque può effettuare una review di tale codice, e difatti i progetti sono spesso portati avanti dalle community di appassionati, che si occupano di arricchirli e aggiornarli. Sebbene la condivisione di codice nasce come esigenza insieme all'informatica, negli anni 80 si delinea un cambio di tendenza, ovvero la nascita dei così detti software proprietari o **Closed source**. Le aziende in questo senso facevano firmare agli sviluppatori degli **accordi di non divulgazione** sui codici sorgenti, al fine di secretarli. Dopo lo

sviluppo del sistema operativo **Unix**, venduto inizialmente sotto licenza AT&T (azienda proprietaria dei laboratori Bell dove Unix fu creato) ci fu una sostanziale separazione dei due mondi. In questo periodo infatti Richard Stallman creò il progetto **GNU** ( di stampo Free Source, che si basa sull'ideologia di rendere il software completamente libero ad utilizzi e rielaborazioni da chi lo usa), un sistema operativo Unix-like, e Linus Torvalds il suo kernel, **Linux**.
Il sistema operativo **Android** è un progetto Open source basato su Kernel Linux adattato a dispositivi mobile, mentre quello **Apple** è basato su Unix, in una sua versione Closed source.
Open e Closed hanno entrambi dei pro e dei contro quando si parla di sicurezza.

1. **Security through obscurity**: è un termine utilizzato in informatica per definire un livello di sicurezza dovuto all'occultamento di un particolare elemento. Nel caso del Closed source, si potrebbe pensare che nascondere il codice sorgente ad utenti malintenzionati possa in qualche modo renderlo più sicuro, mentre nel caso Open source lasciarlo liberamente consultabile anche ai fini di generazione di **exploit** (codice malevolo creato al fine di sfruttare una o più vulnerabilità presenti nel codice sorgente di un'applicazione).

   Nella realtà dei fatti, esistono diverse tecniche (alcune sulle quali il progetto qui illustrato si basa) che permettono l'estrazione del codice sorgente e l'esfiltrazione di informazioni da applicazioni Closed source, mentre mettere a disposizione il codice sorgente ad una larga comunità di appassionati permette un controllo incrociato sulla sicurezza di quest'ultimo.

2. **Blind trust**: fiducia cieca, è quella che un utente di applicazioni di software

proprietario ha e deve avere nei confronti dell'azienda fornitrice. Non sapere cosa il codice sorgente contenga al suo interno è di per sé una falla di sicurezza, cosa nega ad un'azienda di inserire del codice malevolo nei suoi software se nessuno può controllarlo ?

3. **Proprietary hardware**: il software non è nulla senza hardware, nel caso di Apple, anche gran parte dei suoi componenti hardware sono Closed source, ovvero creati e protetti da segreti aziendali. Proprietary hardware richiede proprietary code.

## 1.2 Code review e installazione di applicazioni di terze parti

Una fase vulnerabile nel ciclo di vita dei dispositivi mobile è sicuramente quello dell'**installazione** di applicazioni. Le applicazioni presenti all'interno di un dispositivo possono essere di due tipi: applicazioni native e applicazioni di terze parti. Le applicazioni native corrispondono a quelle applicazioni create ed inserite nel dispositivo direttamente dalla casa madre che l'ha creato, o che vengono successivamente distribuite sui propri canali ufficiali, sono app solitamente più sicure poichè anche il rilascio di aggiornamenti è regolarmente previsto dall'azienda vendor, in poche parole sono le app che si trovano già presenti all'acquisto di un dispositivo.
Le applicazioni di terze parti sono invece prodotte da fonti terze, non per forza ufficiali e non per forza con un alto livello di credibilità.
Apple in questo senso effettua un controllo mirato utilizzando il suo store applicativo come fosse un vero e proprio bastione difensivo, difatti tutte le app disponibili per sistemi Apple devono passare per quella strada. In questo bastione, prima della pubblicazione effettiva dell'app sullo store, vengono effettuati dei controlli sconosciuti a noi utenti. Tutto

quello che sappiamo è che le linee guida per i developer di applicazioni apple sono molto chiare e stringenti, ma quello che effettivamente accade una volta che l'applicazione è stata mandata al bastione per la review, è a noi negato saperlo.
Visto che la review non comprende l'analisi del codice sorgente ma solamente quella del file binario, viene da pensare che vengano utilizzate delle tecniche simili ( anche se più mirate e complesse ) a quelle spiegate nei capitoli che seguono in questo documento.
Di suo conto invece, un utente Android può decidere se installare delle applicazioni da fonti ufficiali come il **Google play store** o altri canali riconosciuti dalla community.
Questo espone un utente medio ad una serie di rischi incontrollati, dovuti all'importazione di codice sorgente spesso non verificato sul proprio device.
Applicazioni malevole contengono codice solitamente ben nascosto in grado di "risvegliarsi" una volta avviate.

## 1.3 Sandboxing e gestione dei permessi

Nei dispositivi Apple tutte le applicazioni, comprese quelle di terze parti sono eseguite in ambiente Sandbox. Con questo termine si definisce la creazione di un perimetro attorno all'applicazione (come una scatola), che le permette di effettuare solamente delle precise azioni. In questo modo dunque Apple definisce dei percorsi specifici dove i file di quella applicazione dovranno essere inseriti, e ogni applicazione riceverà questi indirizzi casualmente. Inoltre ogni applicazione potrà usufruire solamente dei permessi sul sistema definiti in determinati file di configurazione che verranno (con ottime probabilità) revisionati durante la code review. Ad esempio, un'applicazione potrebbe aver bisogno di utilizzare il microfono o la telecamera del dispositivo, ma lo potrà fare solo nelle modalità specificate e previa autorizzazione dell'utente al momento del bisogno.

Ogni applicazione potrà usufruire dei suoi permessi utilizzando delle API di sistema, funzioni scritte da Apple appositamente per lo scopo, garantendo che non ci siano comportamenti imprevisti. In generale, le applicazioni Ios richiedono all'utente l'approvazione di alcuni permessi al loro primo utilizzo, e gli può venire revocato il permesso in ogni momento, ma i permessi a disposizione sono uguali per tutte le applicazioni. Nonostante tutto questo non abbia risparmiato i dispositivi Apple dall'essere violati nella storia recente, fornisce sicuramente un ottimo layer di sicurezza.

In ambiente Android ci sono delle differenze, anche qui tutte le applicazioni vivono in

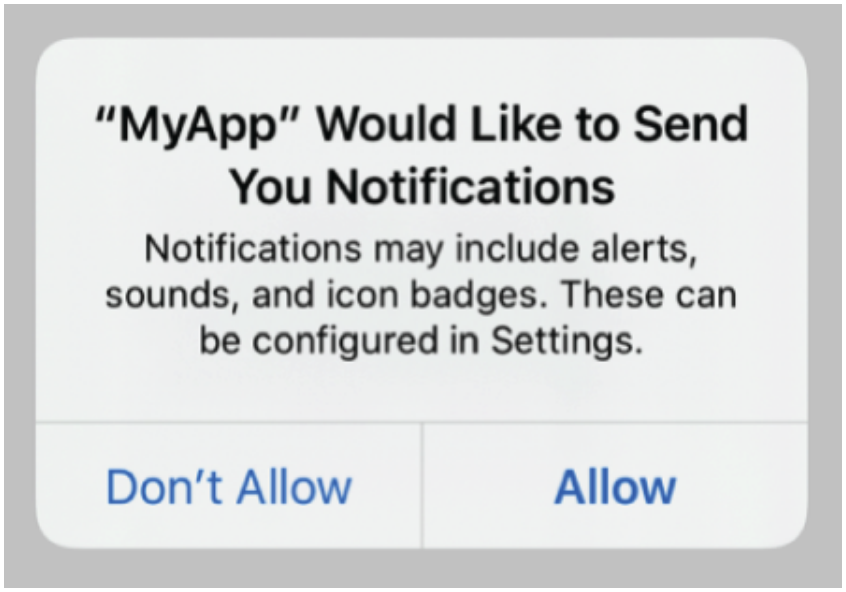



Figura 1.1: Esempio di richiesta permessi su Ios

un Sandbox enviroment, ma innanzitutto i permessi richiesti dall'app vengono accettati tutti insieme dall'utente al momento dell'installazione, con conseguente rischio per un utente poco attento di accettare qualcosa di nocivo. Un secondo aspetto importante è che le applicazioni possono utilizzare molte più chiamate di sistema fornite dal sistema operativo sottostante, e non solo alcune. Questo, in caso di presenza di codice malevolo passato inosservato all'utente, può rischiare di dare troppo potere a delle applicazioni

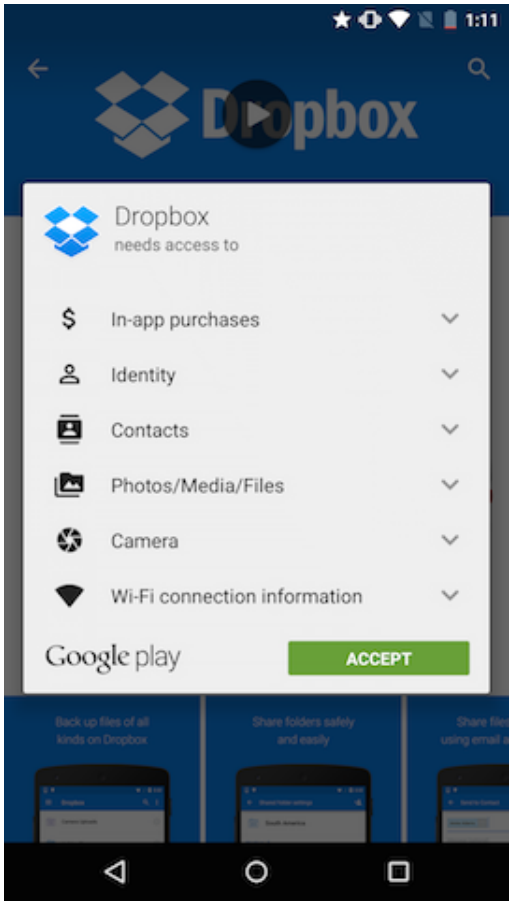



Figura 1.2: Esempio di richiesta permessi su Android al momento dell'installazione

che non dovrebbero averne. Inoltre, è possibile "uscire" dall'ambiente sandboxing manomettendo alcune componenti del Kernel Linux. Al netto di questo e altre informazioni più tecniche, l'hardening del sandbox enviroment sembra essere più stringente nei sistemi operativi Ios rispetto a quelli Android.

## 1.4 Jailbreak e rooting

È chiaro come le applicazioni, così come gli utenti, vengano mantenute appositamente in ambienti controllati al fine di garantire un controllo adeguato ed evitare cattive sorprese,

eppure negli anni sono state scoperte delle vulnerabilità che permettevano a determinate applicazioni o utenti malintenzionati (eccezion fatta per i ricercatori) di rompere questa "gabbia" costruita intorno a loro e riuscire ad usufruire a pieno della potenza del sistema operativo presente sul dispositivo.

Nel caso di Apple, queste tecniche prendono il nome di **Jailbreak**, proprio per definire la rottura della "gabbia" in cui le applicazioni vengono confinate e controllate.

Il Jailbreak fa in modo che un utente possa avere molto più potere sul software presente sul dispositivo, e accedere a funzionalità non previste dall'azienda di produzione, per fare un esempio, un telefono Apple che ha subito Jailbreak, potrà installare software di terze parti bypassando i controlli dell'App store, quindi inserendo software personalizzato all'interno del dispositivo.

Possedere un telefono che ha subito Jailbreak ci porta davanti a una serie di vulnerabilità critiche per l'utente. Innanzitutto come già detto, l'installazione di applicazioni non adeguatamente certificate può veicolare virus e malware dentro il dispositivo. Sebbene molte app di private internet banking siano progettate affinchè non possano essere installate su telefoni che hanno subito Jailbreak (perché i dati dell'utenti non sarebbero più protetti dai meccanismi Apple), effettuare questa scoperta risulta a volte impossibile.

Un telefono che ha subito Jailbreak a tutti gli effetti infatti potrebbe non mostrare differenze grafiche, né di sistema, ma ci si può rendere conto se sul telefono siano installate applicazioni non presenti sull'app store. Utilizzare questa tecnica quindi ci darà sicuramente un maggiore potere sul nostro dispositivo, ma ci esporrà esponenzialmente ai rischi provenienti dall'esterno.

Nei dispositivi Android invece si parla di **rooting**, il termine deriva da **root**, che nel sistema Linux rappresenta l'utente più importante del sistema.

A differenza del jailbreak, il rooting è più potente in quanto mette in mano all'utente

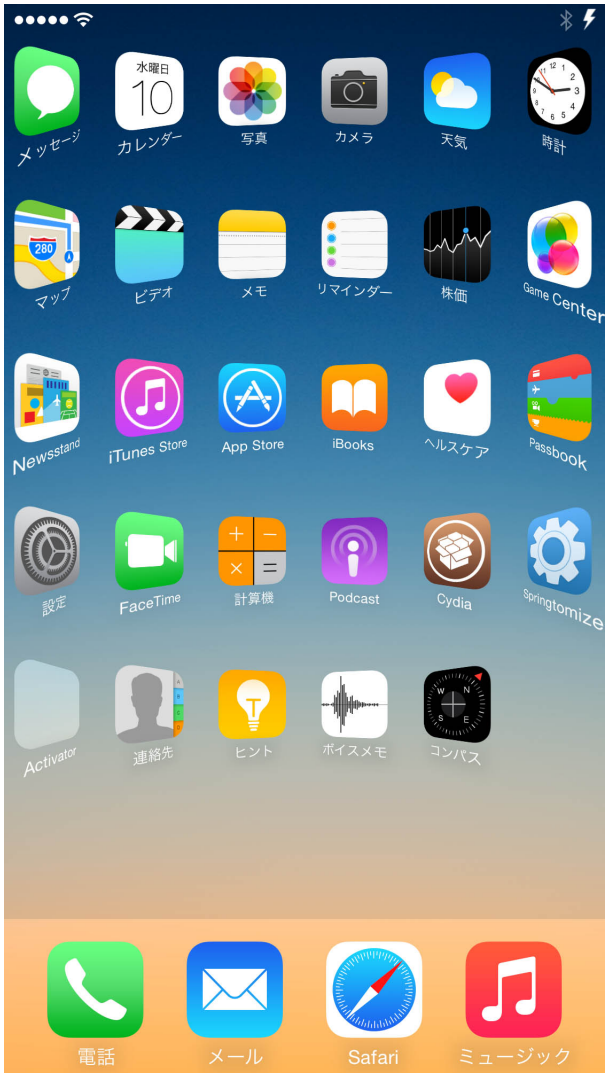



Figura 1.3: Un **Iphone** che ha subito **jailbreak**, il layout è cambiato e si può notare la presenza di **Cydia**, uno store alternativo che permette l'installazione di app di terze parti

tutto il potere del sistema operativo. Se per esempio su un dispositivo Apple poter installare app di terze parti è di per se una rivoluzione, su Android questo è già possibile, dunque il rooting viene utilizzato per molti altri scopi. Inutile dire che un telefono sottoposto a rooting, nel caso fosse infetto da malware, non avrebbe scampo e che tutti i dati dell'utente presenti sulla memoria del telefono sarebbero a rischio di essere letti, rubati,

cancellati o modificati.

## 1.5 Struttura dei pacchetti applicativi

Le applicazioni che installiamo ogni giorno sui nostri telefoni sono composte da decine se non centinaia di file.
Nel mondo Apple l'estensione di questi pacchetti si chiama **Ipa** (Iphone Application). Questa estensione rappresenta un archivio zip con una struttura gerarchica dei file al suo interno ben definita [6]:

- /Payload/
- /Payload/Application.app/
- /iTunesArtwork/
- /iTunesArtwork@2x/
- /iTunesMetadata.plist
- /WatchKitSupport/WK
- /META-INF/

La cartella **Payload** è la più importante ed è quella dove sono inseriti i file utili al funzionamento vero e proprio dell'applicazione, come file di configurazione (i famosi file con estensione **plist**), il file binario eseguibile vero e proprio (in formato Mach-0) ,le librerie necessarie e quelle per il supporto multi linguaggio. La cartella META-INF contiene solo informazioni su quale programma è stato utilizzato per creare l'applicazione, mentre quelle di iTunes contengono file multimediali e immagini.

Nei dispositivi Android invece l'estensione dei pacchetti è **Apk** (Android package). Anche questa estensione rappresenta un archivio con questa gerarchia: [3]:

- /META-INF/
- /lib/
- /res/
- /assets/
- AndroidManifest.xml
- classes.dex
- resources.arsc

La cartella **META-INF** contiene le informazioni incluse nell'applicazione al momento del suo rilascio, al suo interno troviamo file come Manifest.mf che contiene un digest SHA-256 di tutti i file presenti nell'applicazione per verificarne la loro veridicità, questo file và aggiornato ogni qualvolta l'applicazione viene modificata. La cartella **lib** contiene tutte le librerie neccessarie al funzionamento dell'app su diversi processori. Nella folder **res** troviamo dei file dedicati ai settaggi dell'app sul dispositivo (grandezza schermo, display ecc..), simile alla cartella **assets** con la differenza che quest'ultima è gestita dallo sviluppatore che potrà riempirla con tutte le cose necessarie all'applicazione, come documenti, templates e anche del codice. Il file **AndroidManifest.xml** è simile ai file **plist** di Apple, qui infatti troviamo tantissime utili informazioni per riconoscere l'applicazione, quindi i suoi componenti chiave. Più tardi nel documento verrà chiarita l'importanta di questi file di configurazione. **Classes.dex** contiene del codice compilato, mentre **resources.arsc** informazioni per utilizzarlo.

## 1.6 Vulnerabilità dei file di configurazione

I file di configurazione delle applicazioni Android, come Apple, possono essere dei veri e propri tesori per utenti malintenzionati.
In questi file vengono orchestrati i permessi, le azioni e le dipendenze delle applicazioni, nonchè il loro rapporto con alcuni elementi del sistema del dispositivo. Di seguito si elencano alcune informazioni presenti in questi file che potrebbero rendere l'applicazione vulnerabile :

- Presenza della flag di **debug**: la presenza di questa flag consente ad un utente malitenzionato di "debuggare" l'applicazione, quindi effettuare tramite altri programmi un'analisi di tutti i suoi movimenti, le azioni sui dati, e notare informazioni che non dovrebbero poter essere visibili a terzi.

- Credenziali cablate: molto spesso chi sviluppa delle applicazioni, per comodità o ingenuità, si avvale dei file di configurazione per l'inserimento di credenziali che dovrebbero rimanere segrete. Se non viene posta attenzione a questo processo e le credenziali rimangono nei file anche dopo il rilascio dell'applicazione, queste saranno completamente visibili a occhio nudo con pochissimo sforzo da utenti malintenzionati.
  Seppur banale, questo problema è molto più comune di quanto si immagini.

- Presenza della flag di **backup**: un utente malintenzionato in possesso del vostro dispositivo mobile potrebbe sfruttare questa vulnerabilità per acquisire dati protetti dall'applicazione, ottenendo informazioni fornitegli direttamente dall'applicazione tramite backup, ovvero salvataggio dei dati in un preciso momento temporale.

- Permessi: nei file di configurazione sono presenti i permessi che l'applicazione richiederà al sistema operativo per poter svolgere le sue funzioni base, questi potrebbero venir alterati, consentendo all'applicazione di avere più potere di quanto dovrebbe (vulnerabilità che riguarda specialmente i dispositivi Android).

## 1.7 Note conclusive

Scopo di questo capitolo è stato dare al lettore una piccola e mirata overview per essere in grado di intendere il processo di sviluppo del progetto trattato di seguito. Questo documento non ha la pretesa di esplicare in maniera esauriente tutte le differenze anche in ambito di sicurezza che differenziano il mondo Apple dal mondo Android.

# Capitolo 2

# Analisi Progettuale

È possibile realizzare un programma, che data un'applicazione mobile in fase di rilascio sul mercato, vada ad analizzare la presenza o meno di vulnerabilità di diversi tipi, e decida di sospendere o meno il suo rilascio a seconda dei casi ?

Questa è stata la prima domanda alla base del progetto, il suo scopo primario.

Subito dopo mi sono chiesto, chi dovrebbe essere l'utente finale di questo progetto ? Per chi questo progetto dovrebbe essere pensato ?

Ragionare un programma per un uso specializzato dunque rivolto ad una ristretta cerchia di utenti è diverso dal ragionarne un altro dedicato ad un bacino di utenza più ampio, dunque a personale non per forza specializzato. La scelta è stata guardare perciò alla creazione del programma finale in un'ottica di pipeline aziendale, ovvero in grado di far parte del ciclo di vita di sviluppo di un software, precisamente nella parte ultima, precedente al rilascio.

In altre parole quello che ho voluto costruire è stato un programma che potesse avere l'ultima parola sul rilascio nel mercato (nello specifico il mercato Apple) di un'applicazione, che quindi agisse come giudice, approvandola o respingendola, dopo averla accuratamente analizzata secondo delle "regole di sicurezza" specifiche. Il programma quindi

avrebbe lavorato su un'applicazione finita, già utilizzabile dall'utente finale, pronta ad essere esposta ai rischi del mondo esterno.
Ma cosa si intende con applicazione *finita* ?

### 2.0.1 Ciclo di vita del software

Il Software Development Life Cycle, o SDLC [11] [10] [1], è un framework che definisce l'insieme delle azioni che vengono effettuate ad ogni passo del ciclo di vita di un software. È composto da 7 fasi principali.

1. **Pianificazione**: la fase dove si decide lo scopo e il proposito dell'applicazione, in questa fase i capi progetto, insieme al team ma anche ai clienti, decidono come impostare il lavoro, definendo delle scadenze, goal e costi.

2. **Analisi e definizione dei requisiti**:questa fase serve per capire cosa e quali risorse serviranno veramente per lo sviluppo.

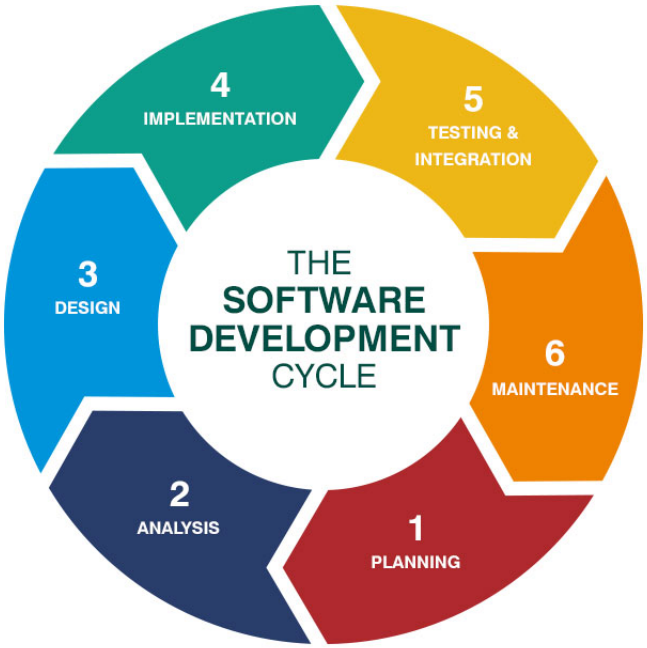
3. **Design**: la fase dove l'architettura del progetto viene pensata e resa possibile rispettando i requisiti.

4. **Sviluppo del software**: in questa fase avviene l'effettiva scrittura del codice.

5. **Testing**: il prodotto inizia ad essere testato, i test possono avvenire manualmente, con test automatici o UAT (User Acceptance Testing).

6. **Rilascio e manutenzione**: questa è la fase finale in cui il prodotto inizia ad essere utilizzato e quindi mantenuto nel tempo.

Il progetto qui spiegato vorrebbe collocarsi quindi tra la fase di **testing** e la fase di **rilascio**, prendendo come target dunque un'applicazione per dispositivi mobile pronta all'uso.




## 2.1 Overview del progetto

Il progetto prende il nome da **Medusa**, in onore della figura mitologica greca (Médousa, che vuol dire "protettrice", "guardiana", da médō, "proteggere"), è infatti l'idea che stà alla base del programma quella di agire da guardiano e proteggere l'applicazione da errori o disattenzioni commessi nella fase di sviluppo, dove nascono i problemi più infimi relativi alla sicurezza.
È stato simpatico anche pensare che Medusa potesse pietrificare il rilascio dell'applicazione, evitando che fosse esternata sul mercato.

La prima domanda è stata, come affontare l'analisi delle vulnerabilità di un'applicazione completa, pronta ad essere rilasciata sul mercato ?
C'è da dire infatti che essere in possesso del codice di un'applicazione consente all'anali-

sta di sicurezza di effettuare controlli su vulnerabilità note con più semplicità, ma cosa fare quando il codice non è presente e l'applicazione da analizzare viene fornita nella sua ultima fase, ovvero già compilata e pronta all'uso ? È per rispondere a questa domanda che il workflow del programma è impostato in modo che potesse automatizzare delle tecniche di **reverse engineering** [7] e **malware analysis** [15]. Tramite l'utilizzo di queste tecniche è stato possibile estrarre le informazioni necessarie e farle elaborare al programma così da avere un responso.

### 2.1.1 Reverse engineering e malware analysis

L'ingegneria inversa, o reverse engineering, è un insieme di tecniche atte a fornire conoscenze tecniche e informazioni spesso riservate, che quindi non dovrebbero essere fruibili alla normale utenza, di un prodotto pronto all'uso e quindi non più in fase di costruzione. L'alba di questa tecnica così definita sembra risalire alla prima rivoluzione industriale, è qui infatti che per scopi commerciali si iniziò a sentire il bisogno di indagare su come alcuni prodotti fossero costruiti in modo da emularli. A dire il vero questa tecnica risiede nello spirito umano fin dalla sua nascita, è generata da domande del tipo, com'è costruito questo oggetto, cosa gli permette di fare questo, di cosa e da cosa è composto ?
La reverse engineering permette, tramite l'utilizzo di svariati metodi a seconda del campo di applicazione, di ottenere informazioni anche volutamente nascoste dal produttore di un oggetto, ai fini di ricerca o spionaggio industriale.
Nel settore informatico, la reverse engineering è molto usata per analizzare software proprietari e non Open source.
Per quanto riguarda il software abbiamo due modi in cui esso può venire analizzato, la prima e più comune modalità statica, e la seconda più avanzata che è quella dinamica. La modalità statica comprende l'analisi di un programma senza che esso venga effetti-

vamente avviato.

In questa fase si cerca innanzitutto di estrarre più informazioni possibili riguardante il software in analisi, lo si fa guardandolo sotto diversi punti di vista, facendo ipotesi sul suo comportamento, sulla tecnologia che lo compone, e provando ad estrarre quello che viene chiamato codice **assembly**, ovvero codice di basso livello che descrive le istruzioni che il processore dovrebbe eseguire una volta lanciato il programma. Non essendo possibile leggere il codice originale, l'assembly fornisce ( per chi lo sà interpretare) una visione completa, seppur molto più complessa da analizzare, delle azioni svolte dal programma. In origine queste cose venivano fatte a mano, ma oggi è possibile usufruire liberamente di software in grado di avvelocizzare il processo, come *debugger* e *decompiler*.

Durante l'analisi statica quindi si acquisiscono tutte le informazioni sul comportamento che il programma potrebbe avere una volta avviato. Durante l'analisi dinamica invece, il programma viene sottoposto a stress e tramite vari strumenti come possono essere analizzatori di rete o programmi che monitorano le chiamate di sistema, si cerca di avere un quadro chiaro del suo comportamento, questi test vengono normalmente fatti in ambienti protetti per evitare sorprese sgradite (si pensi all'analisi di un software potenzialmente pericoloso).

Nonostante esistano modalità di lavoro collaudate negli anni, l'ingegneria inversa resta di per sé un processo molto personale e personalizzabile.

La reverse engineering è di fatto una tecnica usatissima nel campo della malware analysis, che altro non è che una branca della sicurezza informatica dedicata allo studio e appunto all'analisi di software malevoli che vivono sotto l'ombrello del termine **malware** [4].

Non essendo stato in grado di poter **eseguire** le applicazioni sotto esamina per motivi tecnici (mancanza di un dispositivo Apple), quello che mi rimaneva da fare era analizzare

i file presenti all'interno con l'aiuto di questo sistema.
Essendo il programma pensato per lavorare con applicazioni di terzi, è stata palese la necessità di generare della documentazione, o meglio una sorta di report sull'analisi effettuata dal programma, al fine di creare uno storico e rendere chiaro il perché di un eventuale risultato. Il progetto è stato così diviso in tre fasi principali.

1. Estrazione di metadati

2. Analisi dei file e creazione della documentazione

3. Fase decisionale

## 2.2 Estrazione di metadati

Con metadati [5] intendiamo informazioni che descrivono dati appartenenti a determinati insiemi, un esempio molto comune sono le informazioni riguardanti la grandezza di un'immagine, il giorno e l'ora in cui è stata prodotta, e così via. Estrarre i metadati da un'applicazione significa dunque venire a conoscenza di quali sono i suoi contenuti, quanti sono e come sono organizzati.
Essendo l'idea quella di creare un vero e proprio report per l'utente finale, l'estrazione dei metadati è un processo che comincia e perdura per tutto lo svolgimento del programma, il documento infatti viene creato e aggiornato via via che altre informazioni vengono scoperte.

## 2.3 Analisi dei file presenti nell'applicazione

Prima ho parlato di regole, e infatti l'idea di base è stata quella di creare un programma **modulare** che usufruisse di regole scritte ad hoc nella ricerca di vulnerabilità. La modularità qui intesa è quella ottenibile permettendo a qualsiasi utente di poter cancellare, aggiungere o modificare le regole di cui il programma si ciba, lasciandogli la libertà di ottimizzare e affinare la ricerca su vulnerabilità specifiche che gli interessano, oppure effettuare una scansione totale dell'applicazione sotto esamina.
Questa fase si pone il compito quindi di effettuare una analisi di tutti i file presenti all'interno delle cartelle dell'applicazione tramite l'utilizzo di queste regole e non solo, le modalità utilizzate nello specifico verranno trattate nel capitolo relativo all'implementazione.
La domanda seguente è stata: come si può creare un sistema in grado di "lanciare" queste regole sui file e determinare se una vulnerabilità è stata scoperta o meno ?
La risposta è stato un software chiamato **Yara**.

### 2.3.1 Yara

Nel mondo della malware analysis and detection, Yara rappresenta uno standard de facto.
Questo software è un **motore di regole** [13], ovvero un programma che fornisce il modo, tramite appunto delle regole di cui parlavamo anche prima, di analizzare un elemento e sapere se tale elemento "rispetti" o meno quelle regole.
Yara ha un suo preciso formato sintattico ed è senza dubbio un programma modulare, preciso per il mio scopo.
Una regola utilizzata da Yara si compone di diversi elementi, innanzitutto ci sono dei

**pattern** [8], che sono quelli che Yara ricercherà all'interno del file in questione. Questi pattern possono essere stringhe di testo, espressioni regolari, numeri o sequenze di bytes. In particolare è importante notare che le espressioni regolari permettono la creazione di pattern anche complessi in grado di soddisfare altrettanto complesse ricerche.
Un altro elemento presente in una regola Yara è la condizione. Una condizione è un singolo o insieme di valori booleani che Yara confronterà sul file. Un esempio può essere la condizione che uno solo dei parametri venga trovato, piuttosto che entrambi.

```
rule inject_thread {
    meta:
        author = "x0r"
        description = "Code injection with CreateRemoteThread in a remote process"
        version = "0.1"
    strings:
        $c1 = "OpenProcess"
        $c2 = "VirtualAllocEx"
        $c3 = "NtWriteVirtualMemory"
        $c4 = "WriteProcessMemory"
        $c5 = "CreateRemoteThread"
        $c6 = "CreateThread"
        $c7 = "OpenProcess"
    condition:
        $c1 and $c2 and ( $c3 or $c4 ) and ( $c5 or $c6 or $c7 )
}
```

Figura 2.1: Esempio di regola

Sfruttando la potenza di Yara è stato possibile sopperire ai bisogni del programma . Se però Yara è un software utilizzato principalmente per la ricerca di malware signatures, ovvero sequenze di bytes che li rappresenta come il dna rappresenta un essere vivente, ho pensato potesse aiutarmi anche a cercare delle vulnerabilità non esattamente nel suo mirino.
Una delle vulnerabilità più banale ma anche più pericolosa in questione è quella delle credenziali lasciate allo scoperto nei file di configurazione [14], quindi alla mercè di un ipotetico attaccante. Il rischio di incorrere in password e dati di login cablati nei file in analisi non è da mettere in secondo piano.

Grazie a Yara, è stato possibile creare delle regole ad hoc per questi tipi di problemi, riuscendo a provare la modularità del programma modificandole, aggiungendone delle nuove o cancellandole.

## 2.4 Fase decisionale

Arrivati a questa fase la domanda è stata: come fare in modo che il programma possa decidere, a seconda della gravità delle vulnerabilità riscontrate nell'applicazione, se continuare con il rilascio di quest'ultima o agire diversamente (ad esempio bloccandola completamente) ?
Anche nel rispondere a questa domanda, è stato decisiva la volontà di rimanere ancorato all'idea iniziale di modularità, sebbene non sia difficile per un programmatore far sì che una macchina prenda delle decisioni seguendo delle condizioni date, quello di cui c'era bisogno era un modo per permettere ad un qualsiasi utente di poter personalizzare queste decisioni e fare in modo che in ogni momento potesse aggiungerne delle altre o modificare quelle già implementate.
La soluzione decisa è stata quella fornita da **Drools** [9].

### 2.4.1 Drools

Questo software sponsorizzato da Red Hat non è niente meno che un Business Rules Management System (BRMS), ovvero un sistema che permette di gestire delle azioni condizionate a precise regole, quindi collegare causa a conseguenza.
Con Drools è possibile specificare un'azione corrispondente alla veridicità o meno di una condizione. Differentemente da Yara, Drools accetta più di un formato sintattico per definire queste sue regole, che si prensentano complete della coppia condizione-azione .

È infatti disponibile un ulteriore metodo a quello scritto .

Drools offre quella che viene chiamata **decision table**. Quest'ultima è una tabella excel, con delle colonne rappresentanti le condizioni e le azioni da compiere.

Le righe invece costituiscono un po' la linea temporale dello svolgimento, infatti partendo da sinistra e arrivando a destra, una condizione viene verificata, e se risulta essere vera, allora si procede verso l'azione corrispondente, altrimenti ci si blocca.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | **RuleSet** | rules | | | |
| 2 | **Import** | com.cloud.cost.model.BlueReport | | | |
| 3 | **Sequential** | TRUE | | | |
| 4 | **Notes** | This DT translates Channel name to Originating Country and Commpa | | | |
| 5 | | | | | |
| 6 | RuleTable Department to Originating Country | | | | |
| 7 | NAME | CONDITION | ACTION | ACTION | ACTION |
| 8 | | blueReport : BlueReport | | | |
| 9 | | channel_name==$param | blueReport.setCountry($param); | blueReport.setOrg_co($param); | blueReport.appendRuleAudit($param); |
| 10 | **NAME** | **Channel Name** | **Set Orig Country** | **Set Orig Company** | **Set IBM Orig Country Name** |
| 11 | DIV 07 | "Internal US Division 07" | "929" | "BLUEC" | drools.getRule().getName() |
| 12 | DIV 02 | Internal EU Division 02" | "482" | "SOFTC" | drools.getRule().getName() |
| 13 | DIV N95 | "Internal EU Division N95" | "013" | "SOFTC" | drools.getRule().getName() |
| 14 | Div 8M | "Internal US Division 8M" | "710" | "SOFTC" | drools.getRule().getName() |
| 15 | ALL OTHERS | All Other values... | "999" | "SOFTC" | drools.getRule().getName() |
| 16 | | | | | |
| 17 | | | | | |

Figura 2.2: Drools decision table

Grazie a questo accessorio (la tabella), è stato possibile far persistere la modularità del progetto e permettere ad utenti anche non specializzati di cambiare o aggiungere in corso d'opera regole decisionali.

Riassumendo dunque, il progetto si articola in 3 fasi principali, estrazione di dati sull'applicazione e creazione di un documento con funzione di report, controllo delle regole di sicurezza e quindi health check, fase decisionale in cui è possibile decidere cosa fare a seconda dei casi.

# Capitolo 3

# Implementazione

Una volta definite le linee guida del progetto, è iniziata la cernita effettiva delle tecnologie da utilizzare e la conseguente scrittura del codice.
Lo spazio di lavoro è stato **Visual Studio Code** utilizzato in ambiente **Linux**.
La prima scelta implementativa è stata quella di utilizzare il linguaggio di programmazione **Java**, questo perché non solo è uno dei linguaggi più potenti e famosi per la creazione di applicazioni desktop, ma anche perché è supportato dalla maggioranza (se non tutti) i sistemi operativi.
Java venne creato con il motto "write once, run anywhere", è infatti possibile eseguire codice grazie all'installazione della Java Virtual Machine sulla macchina target. A differenza di altri linguaggi di programmazione infatti, i programmi scritti in Java vengono tradotti in Byte Code, uno speciale tipo di linguaggio a basso livello che verrà eseguito dalla JVM.
Questo permette a Java di essere indipendente dal contesto di sviluppo e il pregio di essere Cross-Platform (multipiattaforma), termine utilizzato per specificare applicazioni e/o codice eseguibile su più piattaforme o sistemi operativi.
**Medusa** vuole essere un programma per command line, utilizzabile tramite **shell**, que-

sta scelta è stata arbitraria e dettata oltre che da una passione dell'autore per l'utilizzo del terminale anche da una volontà di rendere il programma di rapido utilizzo senza la "pesantezza" di librerie grafiche, và inoltre detto che non sempre si dispone dell'utilizzo della grafica in alcuni ambienti server, si veda l'esempio di una connessione da remoto, in questo caso rimarrebbe comunque possibile utilizzare il programma dalla shell di servizio.

Il concept di **Medusa** vuole essere quello di un'applicazione già pronta per l'uso, senza bisogno di installazioni aggiuntive, e che abbia meno requisiti minimi possibili. Per fare questo è stato utilizzato **Apache Maven**, un sistema di gestione di progetti basato su Java. Maven permette il rilascio di programmi con installate all'interno tutte le dipendenze necessarie al loro funzionamento.

### 3.0.1 Criticità nella scelta del linguaggio

La scelta di Java è stata decisiva anche per superare uno scoglio implementativo non da poco, ovvero l'utilizzo del sistema **Drools**. Infatti durante lo studio preparativo del progetto, è venuto fuori come ad oggi non esistano modi più semplici di quelli forniti dalle librerie di Java per l'utilizzo di quest'ultimo. La stessa documentazione ufficiale di Drools basa i suoi esempi su tale linguaggio di programmazione, dunque il progetto è stato un po' forzato ad andare in questa direzione.

### 3.0.2 La questione Yara

Durante l'implementazione un altro scoglio importante è stato quello di gestire l'utilizzo di **Yara** a livello di libreria. Se infatti sarebbe stato possibile utilizzare Yara anche da command line semplicemente facendo eseguire al programma delle chiamate di sistema (quindi simulare tramite chiamate al sistema operativo l'utilizzo del terminale da parte

del nostro programma), non mi è sembrata la soluzione più maneggevole, soprattutto in ottica cross-platform, dove avrei dovuto quindi specificare per ogni sistema operativo le operazioni da compiere.
Dalla documentazione ufficiale di Yara si evince come esistano delle librerie per linguaggi come Python e C, ma non per Java. Ho dunque cercato (ma senza successo) di trovare un escamotage al problema, per poi finire a pensare che sarebbe stato meglio utilizzare Python. Ma come utilizzare un linguaggio dentro un altro ?

### Python insieme a Java

Se l'idea iniziale era quella di creare un programma che potesse avere tutte le dipendenze installate e quindi fosse indipendente dal contesto di utilizzo, l'aggiunta di Python avrebbe destabilizzato questo equilibrio.
Fortunatamente la ricerca di una soluzione è stata alquanto rapida, con **Pynstaller** è infatti possibile impacchettare del codice python con le sue relative dipendenze (esattamente come Maven) in un formato binario, eseguibile indipendentemente in ogni contesto, anche in totale assenza di pyhton sulla macchina.
Va detto però che il formato eseguibile generato varia a seconda del sistema operativo su cui Pynstaller viene utilizzato, non và specificato ed è automatico, questo significa che prima di poter utilizzare Medusa su un nuovo sistema operativo gli script python andrebbero ricompilati.
Grazie a questo sistema è stato possibile fare in modo che Medusa fosse composta principalmente da codice Java ma anche da script python capaci di sopperire alle mancanze di quest'ultimo. Questi script, relativi ad un unico binario, sono fatti partire direttamente da Java tramite una chiamata di sistema (anche qui è comunque c'è stato bisogno differenziare i casi, ma in maniera molto più blanda), creando così un processo figlio che

al termine avrebbe ridato il controllo al processo Java principale.

### 3.0.3 Struttura del programma

Le classi del programma sono divise in 4 macro cartelle che definiscono il loro scopo:

1. **App**
2. **droolsObject**
3. **metaDataParser**
4. **securityChecker**

#### App

Questa cartella contiene l'entry point del programma, nella classe App.java.
L'unico parametro passato a Medusa tramite cmdline rappresenta il path (percorso nel fileystem) dell'applicazione da esaminare, in caso il path fosse errato verrà sollevata un'eccezione. Il flow continua con l'esamina dell'header dell'applicazione fornita, è premura di Medusa infatti constatare che tale applicazione sia del formato desiderato, quindi IPA. Per fare ciò si avvale della lettura di alcuni bytes dell'header del file, esattamente quelli che definiscono la natura del file binario agli occhi del sistema operativo, in questo caso la sequenza è "**0x50 0x4B 0x03 0x04**" come specificato dalle documentazioni. Se il file risulta essere un pacchetto di tipo IPA allora bisogna estrarne i file all'interno, infatti per il sistema operativo questo tipo di binario non è altro che un archivio. La classe App.java istanzierà e chiamerà tutte le altre classi consequenzialmente fino al termine delle operazioni necessarie.

**metaDataParser**

Questa cartella contiene le classi **UnzipFile.java CheckCpu.java** e **MetaParser.java**. **UnzipFile** è la prima classe chiamata da App e il suo ruolo è quello di estrarre, quindi ricreare all'esterno dell'archivio tutti i file presenti al suo interno. Il lavoro è svolto tramite un ciclo che controlla la natura di ogni file ad ogni iterazione e lo ricrea nella destinazione, che in questo caso è stata scelta essere la cartella "results" nella directory di Medusa, questa scelta è stata presa per permettere a chiunque di visionare con mano i file estratti ed evitare problemi di permessi al di fuori della cartella del programma.
Scopo secondario di questa classe è quello di trovare il file Info.plist dove la maggior parte dei metadati dell'applicazione (utili allo scopo) risiedono e sono inseriti automaticamente o manualmente dai programmatori. Essendo possibile (verrebbe da dire quasi sicuro) che all'interno della gerarchia dei file dell'archivio esistano diversi file di nome Info.plist (perché generati automaticamente dal compilatore apple) la ricerca viene effettuata tramite la misura della lunghezza dei path relativi alla cartella "results" dei file di nome Info.plist, terminata la quale il path più corto viene scelto. Sebbene questa ricerca sia un po' fatta " a manoni", grazie alla rigida composizione gerarchica di questi pacchetti sbagliare è praticamente impossibile e in questo modo c'è la certezza di aver preso sempre il file giusto. Il path di questo plist file e anche il path del file ipa originale sono dunque passati alla classe successiva, il **MetaParser**. Come detto poco fa, i parametri che questa classe riceve sono i due path da cui andrà ad estrarre le informazioni necessarie per riempire il file di report. Scopo del parser è infatti quello di analizzare il file plist in cerca di tutte le informazioni più utili, ma anche quello di leggere ulteriormente a fondo il file IPA per estrarre ulteriori informazioni comprese all'interno dei suoi segmenti, per poi inserire tutte nel documento. Per quanto riguarda il file Info.plist, esso

si presenta in formato Xml e per il suo utilizzo da parte di Java è stato utile utilizzare una libreria chiamata **dd**, rilasciata sotto licenza **MIT**. Grazie a **dd** il file viene analizzato e cambiato fino a modificarne il formato in **NSDictionary**, simile all'**HashMap** di Java, in cui successivamente viene trasformato. La classe prosegue con la creazione del file json che servirà da documentazione. Dentro questo file verranno poi inserite tutte le specifiche trovate nel file plist, verranno anche inserite delle informazioni più generiche sull'applicazione, come la grandezza, l'hash md5 e quello sha256. [ inserire foto del report ] Va fatto notare che non tutti i dati si presentano in forma semplice, alcuni sono dizionari oppure array, è stato quindi dovuto specificare il modo in cui estrarre questi dati volta per volta. Durante l'estrazione, è possibile ricavare il nome dell'effettivo binario, che sappiamo anche in questo caso essere quello definito dalla voce "**CFBundleName**" presente nel file Info.plist. Una volta terminata la lettura del file, viene invocata la terza classe, **CheckCpu**, che prenderà come parametro il path del binario vero e proprio e arricchirà ulteriormente la documentazione. In questa classe viene effettuara una lettura ulteriore dell'header del pacchetto Ipa al fine di determinare per che tipo di Cpu è stato compilato, in particolare i binari di tipo Mach-0 utilizzati da Apple presentano un magic number diverso a seconda dell'architettura a 64 o 32 bit. Tutti questi dati vengono poi aggiunti al file "metadata.json".

### securityChecker

In questa cartella sono presenti due classi, la prima, **SecurityChecker.java** è quella che effettivamente si occupa di creare un processo secondario a cui delegherà l'esecuzione degli script python appositamente creati per scansionare tutti i file estratti dall'archivio ipa tramite le regole di **Yara**. Prima dell'esecuzione però, viene effettuato un controllo

per decidere se il sistema operativo in uso sia Windows based o Unix based, questo perché a seconda del caso è necessario utilizzare un programma di esecuzione diverso (**cmd.exe** nel primo caso, **shell** nel secondo). Gli script python sono compilati in un tipo di eseguibile compatibile con il sistema operativo in cui devono essere usati, per questo è anche necessario separare le due esecuzioni.
Gli script python sono 3 al momento della scrittura di questo documento ed eseguiranno in ordine queste funzioni:

1. Il primo utilizzerà una libreria apposita per la lettura di binari di tipo Mach-0 Apple, con lo scopo di ottenere ulteriori informazioni sull'eseguibile che prima non è stato possibile ottenere. Tali informazioni riguardano le flags che il compilatore inserisce nel binario al momento della sua esecuzione, a noi interessano perché la mancanza di alcune delineano una mancanza di perimetro difensivo non da poco. Esempio di alcune flag sono **CANARY**, sistema utilizzato per evitare e registrare l'evento di tentato buffer overflow dell'applicazione. La canary è una tecnica che consente, tramite inserimento di un "canarino" (ovvero una sequenza di byte) subito prima del return address di una funzione, di controllare se esso sia stato o meno cambiato prima che la funzioni appunto ritorni, infatti durante un attaco di tipo buffer overflow classico, lo stack viene corrotto per andare a cambiare il return pointer e prendere il controllo del flow del programma vittima. La presenza di canary nelle applicazioni rende questo lavoro molto più complicato. Un'altra flag molto importante è **PIE**, utile alla randomizzazione di alcuni segmenti di memoria del programma, in modo da rendere tecniche di buffer overflow molto più difficili, con NX abilitato infatti un attaccante farebbe fatica a trovare un indirizzo fisso proprio per via della continua randomizzazione di tali, è simile all'**ASLR**. Un altro esempio di flag di sicurezza è **NX** (Non executable stack) che rende lo stack

e l'heap del programma non eseguibili, ovvero rende impossibile l'inserimento di codice malevolo propriamente inoculato nell'applicazione.

2. Il secondo script avrà lo scopo di passare uno per uno i file della cartella **Payload** estratta dall'archivio ipa, ed eseguire su ognuno di essi tutte le regole Yara. A prima vista questo può sembrare esagerato, e in qualche modo lo è, ma ragionando su applicazioni di private internet banking, la sicurezza non è mai abbastanza, e nonostante questo rallenti notevolmente l'esecuzione al crescere dei file presenti nell'applicazione, Yara è molto agile e veloce e permette di non dover aspettare troppo, avendo la certezza di aver controllato minuziosamente tutto.

3. Il terzo script vuole raccogliere i dati ottenuti dai suoi precedessori e inserirli nel file metadata.json, vengono dunque creati i campi "matches" per le regole di Yara matchate, e "flags".

Dopo l'esecuzione degli script python, il documento di report (metadata.json) è completato, e fornisce una chiara panoramica dell'applicazione e le vulnerabilità riscontrate. Adesso l'ultima classe della cartella, **JsonReader.java** si occuperà di deserializzare i dati di sicurezza ottenuti e popolare l'oggetto Java che poi sarà utilizzato da **Drools**.
Per fare questo effettuerà un parsing del documento e passerà alla classe principale App i risultati, la quale andrà poi a sua volta a passarli alle classi dell'ultima cartella come parametri.

### droolsObject

È l'ultima cartella e contiene le classi utili all'esecuzione del motore **Drools**.
La prima classe **DroolObj.java** è semplicemente un contenitore che viene riempito dalla classe App.java con le informazioni ricavate dal deserializzatore del file json aggiornato

fino a quel momento, nello specifico vengono usate due variabili, la prima è una HashMap contenente Chiave (stringa) e valore(booleano) dedicata alle flags, la seconda è un array dedicato a tutti i matches trovati. La seconda classe **Droolcore.java** si occupa di inizializzare il "server" Drools. Si parla di server, perché Drools è uno dei diversi servizi offerti dal **Kie** server di **Jboss**, è quindi possibile tramite Java istanziare una sua sessione e dopodichè utilizzare le api per eseguire il motore di regole.

Dopo aver inizializzato la sessione, quello che succederà è che il server Drools utilizzerà la decision table da noi creata insieme alle informazioni inserite nelle variabili del Droolobj controllando le condizioni ed eseguendo le azioni corrispondenti.

# Capitolo 4

# Risultati

Il programma **Medusa** è stato sviluppato e testato in ambiente **Unix**, sebbene sia stato progettato con una politica cross-platform, la maggioranza dei test sono stati effettuati su macchine **Unix**, mentre un solo test dedicato al collaudo è stato effettuato in ambiente **Windows 10**.

Nonostante il progetto sia dedicato principalmente all'analisi di applicazioni di **internet banking**, è stato possibile analizzare solamente un'applicazione di questo tipo, per ovvi motivi di privacy e permessi di sicurezza legati alla sensibilità della tematica .

L'applicazione è stata gentilmente fornita dall'azienda di consulenza informatica **Imola Informatica**, e risulta essere un progetto di private internet banking in fase di rilascio.

Tutte le altre applicazioni esaminate durante lo svolgimento dei test sono state ricavate da **internet**, specificatamente da repository pubblici, e sono state scelte in modo da poter osservare il comportamento del programma su pacchetti applicativi completamente differenti tra loro .

Sebbene i test siano stati effettuati manualmente senza regole stringenti, i risultati sono stati considerati tenendo conto di alcuni parametri reputati importanti:

1. Presenza di **falsi positivi**

2. **Tempo** di esecuzione

3. **Qualità** della documentazione generata

La presenza di **falsi positivi** [2] è risultata essere uno dei problemi principali durante lo svolgimento dei **test**.
Questo problema è generalmente dovuto alla presenza di regole **Yara** basate su regular expressions ambigue.
Sfortunatamente ad oggi non è possibile dire con certezza che il problema sia stato risolto, sebbene sia stato ridimensionato il più possibile. Tra le applicazioni testate troviamo: **DamnVulnerableIOSApp**, applicazione creata dalla fondazione **Owasp** e disponibile sui loro repository, questa app, come si nota dal nome, è totalmente vulnerabile e quindi rappresenta un ottimo punto di partenza. La stessa applicazione è fornita nei due linguaggi di programmazione utilizzati per la scrittura di applicazioni Apple, **Swift** e **Objective-C** con alcune sottili differenze nei file di configurazione. Un'altra applicazione creata a scopo di ricerca è **iGoat**, anch'essa fornita in entrambe le versioni. Tra le altre applicazioni testate troviamo il famoso gioco **SubwaySurfer** e infine l'app di private internet banking precedentemente citata. Non avendo a disposizione il codice sorgente, la scelta di utilizzare applicazioni volutamente vulnerabili è stata ai fini dei test molto importante per avere un riscontro di veridicità sulle vulnerabilità trovate.

### 4.0.1 Falsi positivi

È stato riscontrato come la curva dei falsi positivi cresca al crescere della grandezza dell'applicazione.
Un falso positivo è spesso dovuto all'utilizzo di caratteri definiti **jolly** o **wild card** nelle regular expressions, essi però sono anche imprescindibili al funzionamento delle stesse,

permettendo infatti di descrivere sequenze di pattern più complessi. Al fine di ridurre questi falsi positivi e identificarli, è stato necessario affinare e rendere più granulari le regole di ricerca delle vulnerabilità, ad esempio spezzettandole in tante piccole sotto regole aventi singoli e più specifici obiettivi.

```
rule admin_pass
{
    strings:
        $a = "admin" nocase ascii wide
        $b = "user" nocase ascii wide
        $c = "(pass|password)" nocase ascii wide
        $e = "^\\S*(passwords?|passwd|pass|pwd)_?(hash)?[0-9]*$" nocase ascii wide
        $ht_passwd = "^htpasswd Hash$" nocase ascii wide
        $nmp_token = "^npm authToken$"
        $pip_passwd = "^pip password$"
    condition:
        any of them
}
```

Figura 4.1: Esempio di regola ambigua

```
rule pip_passwd
{
    strings:
        $pip_passwd = "^pip password$"
    contidion:
        any of them
}

rule password
{
    strings:
        $c = "(pass|password)" nocase ascii wide
        $e = "^\\S*(passwords?|passwd|pass|pwd)_?(hash)?[0-9]*$" nocase ascii wide
    contidion:
        any of them
}
```

Figura 4.2: La regola è divisa in sotto regole

### 4.0.2 Tempo di esecuzione

Il tempo computazionale utilizzato dal programma è $O(n^2)$ . Questo è dovuto alla presenza di un doppio ciclo for innestato negli **script** python, che come spiegato prece-

dentemente, analizzano ogni file dell'applicazione con ogni regola fornita a **Medusa**. Al crescere dei file presenti nell'applicazione presa sotto esamina quindi, il tempo crescerà esponenzialmente.

### 4.0.3 Qualità della documentazione generata

In ogni test effettuato, il livello di dettaglio raggiunto nella documentazione è stato quello sperato, per ogni applicazione è stato possibile estrarre correttamente tutte le informazioni richieste.


```
"Supported Platforms": "(\"iPhoneOS\")\n",
"Min OS Version": {
    "content": "8.0"
},
"Compiler": "com.apple.compilers.llvm.clang.1_0",
"Platform Version ": {
    "content": "15.0"
},
"Size": "162.2MB",
"App type": "Swift",
"Cpu": "arm64",
"SDK name": "iphoneos15.0",
"Name": "SubwaySurf",
"Sha256": "2c2fc4e3a77b0a72bc8ab539be3519673f23c97915c7b183218f167ffc476637",
"Bundle Version": {
    "content": "35036"
},
"Build": "19A339",
"Permissions": {
    "NSAppTransportSecurity": {
        "NSAllowsArbitraryLoads": {
            "integer": false,
            "real": false,
            "boolean": true
        },
```


Figura 4.3: Alcune Informazioni raccolte dall'applicazione SubwaySurfer

### 4.0.4 Risultati più importanti

Di seguito una breve analisi delle vulnerabilità riscontrate da **Medusa** nelle applicazioni testate.

### DVIA, iGoat, SubwaySurfer

Queste applicazioni sono fornite con diverse vulnerabilità, **Medusa** è riuscita a trovarne alcune molto importanti.


```
"matches": [
    "contains_base64",
    "domain",
    "debug_allow",
    "vulnerable_bof_functions",
    "admin_pass",
    "credentials_creditcard",
    "url",
    "IP",
    "Big_Numbers1",
    "BASE64_table"
]
```


Figura 4.4: **DVIA** app


```
"matches": [
    "admin_pass",
    "contains_base64",
    "domain",
    "debug_allow",
    "vulnerable_bof_functions",
    "credentials_creditcard",
    "url",
    "IP",
    "BASE64_table",
    "Misc_Suspicious_Strings",
    "plist_file_vulns"
```


Figura 4.5: **iGoat** app


```
"matches": [
    "debug_allow",
    "contains_base64",
    "url",
    "domain",
    "admin_pass",
    "plist_file_vulns",
    "Big_Numbers1",
    "comments",
    "Misc_Suspicious_Strings"
]
```


Figura 4.6: **SubwaySurfer app**

Si noti dai risultati che nelle prime due applicazioni sono presenti credenziali **amministrative** e numeri di carte di credito, probabilmente di qualche utente. In tutte è presente la regola **debug allow**, che permette ad un attaccante di usufruire della modalità debug di un'applicazione **Ios**. La regola **plist file vulns** specifica che nei file **Info.plist** sono contenute informazioni sensibili o permessi che l'applicazione non dovrebbe avere in fase di rilascio. Le regole **Ip** e **url** testimoniano la presenza di indirizzi ip e url lasciati in chiaro.

### App di private internet banking

Questa applicazione è stato un ottimo caso di studio per verificare il funzionamento di Medusa su una vera applicazione bancaria. I risultati sono stati soddisfacenti.


```
"contains_base64",
"url",
"domain",
"debug_allow",
"vulnerable_bof_functions",
"admin_pass",
"comments",
"credentials_creditcard",
"Misc_Suspicious_Strings",
"IP",
"Big_Numbers1",
"Big_Numbers2",
"Big_Numbers4",
"SHA512_Constants",
"SHA2_BLAKE2_IVs",
"DES_sbox",
"RijnDael_AES_CHAR",
"BASE64_table",
"Str_Win32_Internet_API",
"ldpreload",
"maldoc_getEIP_method_1",
"vmdetect",
"possible_includes_base64_packed_functions",
"plist_file_vulns"
```


Figura 4.7: **Private internet banking app**

Le regole **credentials creditcard**, **admin pass**, **debug allow**, **IP** e **plist file vulns** rappresentano una falla di sicurezza gigantesca per questa applicazione. Và anche notato che Medusa ha riscontrato la presenza di utilizzo di diversi algoritmi di cifratura come **RijnDael_AES_CHAR** e **DES_sbox**.
Mettendoci nell'ottica di un attaccante, tutte queste informazioni vanno a creare un perimetro di attacco molto ampio. A volte anche piccoli commenti possono nascondere falle di sicurezza, preziose informazioni, ma in questo caso sembrerebbero esserci addirittura delle credenziali disponibili ad occhio nudo.

### 4.0.5 Decision table

La decision table di **Drools** è stata implementata per fini dimostrativi in modo da mostrare la coppia causa-conseguenza condizione-azione.

| | | | |
|---|---|---|---|
| **RuleSet** | rules | | |
| **Import** | com.droolsObject.Droolobj | | |
| **Notes** | Decision table for security checks | | |
| RuleTable DiscountCalculation | | | |
| NAME | CONDITION | CONDITION | ACTION |
| | myobj: Droolobj | | |
| | myobj.getSingleFlag($param)==false | myobj.getSingleMatch($param)==true | System.out.println($param); |
| **NAME** | **Flags** | **Matches** | **Deny or Accept** |
| Is Canary enabled | "CANARY" | | "DENY canary is missing" |
| Is Restrict enabled | "RESTRICT" | | "DENY restrict is missing" |
| Is Pie enabled | "PIE" | | "DENY PIE is missing" |
| Is NX enabled | "NX" | | "DENY NX is missing" |
| Is Encrypted enabled | "ENCRYPTED" | | "DENY encrypted is missing" |
| Secrets string found | | "secrets" | "DENY secrets rule matched" |
| contains_base64 found | | "contains_base64" | DENY contains_base64 rule matched |
| domain rule | | "domain" | "DENY domain rule matched" |
| plist_file_vulns rule | | "plist_file_vulns" | "DENY plist_file_vulns rule matched" |
| vulnerable_functions  found | | "vulnerable_functions" | ENY vulnerable_functions rule matche |
| url rule matched | | "url" | "DENY url rule matched" |
| IP rule matched | | "IP" | "DENY IP rule matched" |
| BASE64 table string found | | "BASE64_table" | "DENY BASE64_table rule matched" |
| Misc Suspicious Strings found | | "Misc_Suspicious_Strings" | NY Misc_Suspicious_Strings rule matc |

Figura 4.8: Semplice tabella di decisione

La tabella è collegata all'oggetto Java definito nell'import, da cui prenderà le funzioni richiamate sotto le caselle di condizione. Nelle colonne verdi di sinistra si trovano i nomi delle regole e vulnerabilità trovate da Medusa, divise in "flags" e "matches". Nel caso, e solo nel caso una condizione sia verificata, si procederà all'azione. Nella tabella di esempio (molto semplificata) viene semplicemente stampato un messaggio di errore con una semplice spiegazione del perché non sia possibile procedere al rilascio dell'applicazione sul mercato.

# Capitolo 5

# Conclusioni e sviluppi futuri

Durante lo studio del progetto, c'è stata una fase di test dei prodotti software utili per approcciarsi alla reverse engineering in campo Ios. Un grosso limite è stato quello di non avere a disposizione hardware **Apple**, dovendo quindi scartare a priori situazioni di analisi dinamica che avrebbero permesso una esamina più dettagliata delle applicazioni eseguite. L'inserimento nel progetto di sistemi di analisi dinamica non è al momento contemplato, in quanto accrescerebbe la complessità di quest'ultimo in maniera esponenziale. Medusa è un progetto appena nato, funzionante ma mancante di tantissime features che potrebbero renderlo molto più versatile, di seguito alcune possibili implementazioni future.

### Utilizzo di parametri

Nella sua versione di base Medusa non consente l'utilizzo di parametri per personalizzarne il comportamento, difatti agisce in una unica modalità di analisi.
Alcuni esempi potrebbero essere: specificare file di output, tempo massimo di esecuzione, escludere dei file o fare in modo che solo alcuni file vengano analizzati ecc..

### Multithreading

Medusa è single threaded, questo impatta sulla performance quando si analizzano applicazioni di grandi dimensioni. Un approccio multithreaded potrebbe garantire un alleggerimento nella fase di parsing rendendo il programma più agile.

### Ricerca delle regole

In questo momento la ricerca delle regole per cui è avvenuto il match durante l'analisi è lasciata all'utente, questo risulta essere molto scomodo. Permettere a chi utilizza Medusa di rintracciare la regola e poterla visualizzare semplicemente tramite l'uso di un parametro e il suo nome sarebbe molto utile all'utente.

### Analisi di applicazioni Android

Sebbene Medusa nasca per l'analisi di pacchetti **ipa** destinati al mercato Apple, un'integrazione per l'analisi anche di pacchetti Android la renderebbe ancora più completa.

### Generazione di Sbom per inserimento in pipeline

Medusa è un'applicazione pensata per analizzare un'applicazione subito prima dell'ultima fase del suo life cycle, il rilascio sul mercato. Inserirla in un contesto di pipeline sarebbe possibile e desiderato. Per fare ciò l'idea sarebbe generare un documento Sbom (Software Bill of Materials) permettendo l'utilizzo di Medusa insieme ad altri programmi nella pipeline.

### Interfaccia grafica

Sebbene Medusa sia stata pensata come applicazione per terminale, nel futuro potrebbe essere realizzata una sua versione grafica per facilitarne l'utilizzo su sistemi Windows.

# Bibliografia


[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[2] Onyebuchi Aniekwena. *Identification of Key Factors that Influence False Positive Detection & Classification by Anti-Malware Program*. PhD thesis, Dublin, National College of Ireland, 2020.

[3] Pau Oliva Fora. Beginners guide to reverse engineering android apps. In *RSA conference*, pages 21–22, 2014.

[4] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 2014, 2014.

[5] Jane Greenberg. Metadata extraction and harvesting: A comparison of two automatic metadata generation applications. *Journal of Internet Cataloging*, 6(4):59–82, 2004.

[6] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering ios mobile applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2012.

[7] Hausi A Müller, Jens H Jahnke, Dennis B Smith, Margaret-Anne Storey, Scott R Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 47–60, 2000.

[8] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Kshirasagar Naik, and Jingping Song. Embedding fuzzy rules with yara rules for performance optimisation of malware analysis. In *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–7. IEEE, 2020.

[9] Mark Proctor. Drools: a rule engine for complex event processing. In *International symposium on applications of graph transformations with industrial relevance*, pages 2–2. Springer, 2011.

[10] David F Rico and Hasan H Sayani. Use of agile methods in software engineering education. In *2009 Agile conference*, pages 174–179. IEEE, 2009.

[11] Munish Saini and Kuljit Kaur. A review of open source software development life cycle models. *International Journal of Software Engineering and Its Applications*, 8(3):417–434, 2014.

[12] Fatima Salahdine and Naima Kaabouch. Social engineering attacks: A survey. *Future Internet*, 11(4):89, 2019.

[13] Qin Si, Hui Xu, Ying Tong, Yu Zhou, Jian Liang, Lei Cui, and Zhiyu Hao. Malware detection using automated generation of yara rules on dynamic features. In *International Conference on Science of Cyber Security*, pages 315–330. Springer, 2022.

[14] Erik David Martin1 Iain Sutherland and Joakim Kargaard. Iot security and forensics: A case study. In *ECCWS 2021 20th European Conference on Cyber Warfare and Security*, page 278. Academic Conferences Inter Ltd, 2021.

[15] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.

[16] Enting Zhou, Yurong Liu, Hanjia Lyu, and Jiebo Luo. " ban the chinese spyware!": A fine-grained analysis of public opinion toward chinese technology companies on reddit. *arXiv preprint arXiv:2201.05538*, 2022.

# Ringraziamenti

Se mai dovesse nuovamente insorgere, il buio sul mio cammino, che io possa avervi ancora accanto, pronti a mostrarmi la luce, permettendomi di ritrovare la mia strada.

Gracias a la vida que me ha dado tanto
Me ha dado el sonido y el abecedario
Con el las palabras que pienso y declaro
Madre, amigo, hermano, y luz alumbrando
La ruta del alma del que estoy amando
Mercedes Sosa - Joan Baez