SAPIENZA
UNIVERSITÀ DI ROMA

Automazione dell'analisi statica e dinamica di malware

Facoltà di Ingegneria dell'informazione, informatica e statistica
Laurea in Informatica

Simone Sestito
Matricola 1937764

Responsabile di tirocinio
Prof. Emiliano Casalicchio

Tutor aziendale
Andrea Pedini @ Gyala S.r.l.

Anno Accademico 2022/2023

Automazione dell'analisi statica e dinamica di malware
Sapienza Università di Roma

Questa relazione di tirocinio è stata composta con LaTeX e la classe Sapthesis.

Versione: 5 ottobre 2023

Email dell'autore: sestito.1937764@studenti.uniroma1.it

Abstract

Lo scopo di questo tirocinio è lo sviluppo di un sistema di analisi automatica di eseguibili, sia in ambito Windows che altri sistemi operativi, al fine di comprendere loro caratteristiche e comportamenti. Si predispone per l'integrazione con altri sistemi automatici, quali la Threat Intelligence, seppur sia inizialmente utilizzato dall'analista in modo standalone. Infatti, gli strumenti qui realizzati, possono essere adoperati sia in maniera automatica che manuale per estrarre il maggior numero di informazioni possibili riguardanti l'eseguibile in questione, in maniera isolata e sicura per l'ambiente di lavoro.

Ove possibile, rispettando i termini di servizio delle varie piattaforme, l'analisi viene eseguita in cloud. Altrimenti si usano VM altamente isolate dal sistema host, al fine di avere una sicurezza quanto più elevata possibile, riducendo la superficie di attacco.

I risultati vengono sempre espressi in un formato JSON personalizzato dello strumento creato, sulla base delle esigenze dell'utilizzatore.

Indice

Capitolo 1

Introduzione

1.1 Malware e tipologie

Nell'ambito della sicurezza informatica, un malware è definito come un programma che è inserito in un sistema, di solito di nascosto, con l'intento di compromettere la confidenzialità, l'integrità e la disponibilità dei dati della vittima, o sue applicazione, fino al sistema operativo, o in altro modo infastidire o disturbare la vittima [6].

Possono essere di varie tipologie, o unioni di alcune di esse: [1]

- **Adware**: progettati per presentare messaggi pubblicitari e guadagnare da essi
- **Spyware**: hanno lo scopo di osservare le azioni dell'utente senza il suo permesso per poi riportare il tutto all'autore
- **Virus**: vanno ad alterare altri programmi agganciandoci del proprio codice
- **Worm**: alterano altri computer in una stessa rete, provocando danni
- **Trojan**: ingannano l'utente presentandosi come un software utile, che quindi l'utente va ad installare volontariamente, non sapendo cosa si cela realmente al contrario di ciò che gli è stato promesso
- **Ransomware**: guadagnano sul malcapitato criptando tutti i propri file importanti, chiedendo poi un riscatto in criptovalute per lo sblocco - qui si fa leva sull'importanza e l'assenza di backup di dati importanti e sul proprio valore sia economico che morale (a seconda che si tratti di un'azienda o di un utente domestico)
- **Rootkit**: sfruttano vulnerabilità nel sistema target per ottenere privilegi da amministratore
- **Fileless Malware**: non installa alcunché sulla macchina inzialmente, ma apporta cambiamenti a file nativi del sistema operativo, come qualche DLL essenziale; un esempio è *Astaroth*
- **Keylogger**: monitora l'attività dell'utente, generalmente focalizzandosi sull'input da tastiera, al fine di ottenere tanti dati tra cui le password, i nomi utente e i siti visitati, siccome anch'essi vengono digitati dall'utente; ad esempio

Olympic Vision aveva come target dei businessmen e attaccava le loro caselle di posta elettronica aziendali

- **Bot**: un software installato nel sistema che resta dormiente nell'attesa di intraprendere un'azione generalmente inviata da remoto dal creatore a tutti i dispositivi infettati, che faranno così parte di una botnet
- **Wiper**: differisce dai ransomware per la loro natura di eliminazione dei dati utente, anziché la loro criptazione dietro riscatto, come ad esempio *WhisperGate*
- **Logic Bomb**: un set di istruzioni incluso in un programma, che contiene un payload pericoloso ma che è innescato solo se certe specifiche condizioni logiche sono verificate, come un orario ben preciso o la presenza di software; questo verrà trattato anche nella Sandbox realizzata in merito a pratiche di anti VM detection

Com'è normale pensare, non esistono solo questi tipi, ma sono solo alcuni dei più diffusi e conosciuti.

Infine, per maggiore comprensione delle seguenti illustrazioni, è bene spiegare altri pochi termini:

- Una **vulnerabilità** è una falla o una debolezza nella progettazione, implementazione o gestione di un sistema, che potrebbe essere sfruttato per violare la sicurezza del sistema stesso [5]
- Un **exploit** invece è un software o un insieme di comandi che vanno a sfruttare la vulnerabilità a proprio vantaggio, al fine di provocare un comportamento altrimenti inaspettato
- Si definisce **threat** un pericolo di invasione della sicurezza, che esiste quando c'è una circostanza, una capacità, un'azione o un evento che potrebbe compromettere la sicurezza e causare un danno. In altre parole, è un possibile pericolo che potrebbe essere sfruttata una vulnerabilità. [5]
- Un tipo particolare di exploit chiamato **zero-day** va a sfruttare falle non note prima dell'attacco, provocando così potenzialmente molti più danni, non essendo ancora disponibile una patch del software vulnerabile
- Per **IoC** si intende un Indicator of Compromise, indicatore usato per identificare indirizzi IP o nomi di dominio di C&C, hash di file, firme di antivirus, o altro, che faccia ricondurre con alta probabilità a una specifica intrusione

1.1.1 MITRE ATT&CK

Per identificare più precisamente le tipologie di malware che stiamo trattando, si va ad usare il *MITRE ATT&CK® Framework* (Adversarial Tactics, Techniques, and Common Knowledge) [1].

Si tratta di una base di conoscenza sviluppata da MITRE Corporation, che include tattiche e adversary techniques, basata osservando gli avvenimenti nel mondo

[1] Pagina ufficiale MITRE ATT&CK: `https://attack.mitre.org/`

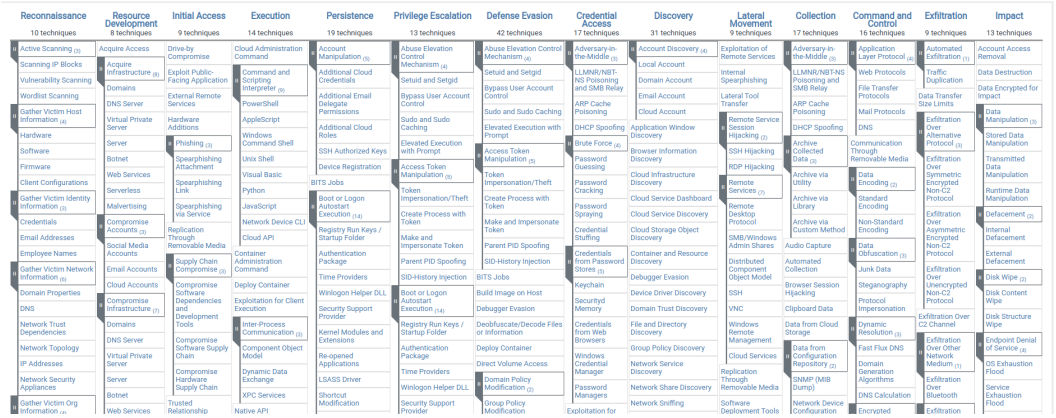


Figura 1.1. MITRE ATT&CK Matrix for Enterprise

reale. Nato nel 2013 con lo scopo di descrivere le più comuni tattiche, tecniche e procedure *(TTP)* usate nei sistemi Windows enterprise, ad uso interno MITRE, è poi diventato lo standard de-facto per la descrizione di tali aspetti di un attacco [7], per far fronte all'assenza di una tassonomia comune per la descrizione dei TTP.

Si focalizza sulle interazioni che il malware ha col sistema target, e raggruppa questi comportamenti in tattiche per fornire più contesto sulla tecnica utilizzata.

- Le **tattiche** sono il *perché* di una tecnica, indicando l'obiettivo che si intende perseguire, e servono da contenitore per le varie tecniche; nello standard iniziano per `TA` (es: `TA0043` - Reconnaissance - Ricognizione, l'avversario va ad ottenere informazioni sul sistema per capire come muoversi)

- Le **tecniche** invece sono il *come* di una specifica azione, e potrebbero far dedurre anche il *cosa* un avversario ottiene come risultato della sua azione - particolarmente utile nella tattica TA0007 Discovery, dove si cerca di studiare la composizione del sistema target.
  Una tecnica è spesso composta da sotto-tecniche per essere ancora più specifici. Ad esempio, la tecnica `T1595.002` Vulnerability Scanning (dove si va capendo i software disponibili sul sistema target e la loro versione, con il possibile scopo di verificare se si allinea a una specifica versione vulnerabile di cui l'avversario già dispone di un exploit) è sotto-tecnica di `T1595` Active Scanning, e fa parte della tattica `TA0043` Reconnaissance vista al punto precedente.

Nella matrice in figura 1.1, le tattiche sono le colonne e le tecniche sono le celle, possibilmente composte da sotto-tecniche.

Viene usato anche per l'integrazione con la Cyber Threat Intelligence, che vedremo nella sezione 1.4, punto focale anche degli strumenti realizzati nel progetto.

1.2 Cyber Kill Chain

Partendo dalle tattiche appena viste, possiamo costruire ciò che è noto col nome di *Cyber Kill Chain*. Si tratta di un modello simile al MITRE ATT&CK framework (sez. 1.1.1), ma segue un diverso approccio. Qui infatti si vanno a ripercorrere le

7 tipiche fasi cronologiche di un attacco ad un sistema informatico, fornendo una panoramica più ad alto livello.

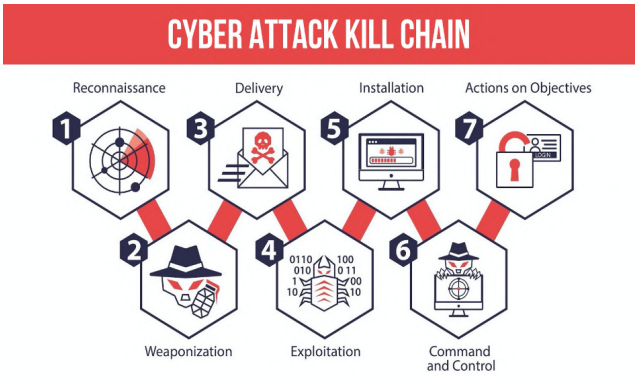


Figura 1.2. (crediti immagine: "Fondazione F3RM1")

Si compone delle seguenti fasi [3], portando un esempio tipico:

1. **Ricognizione**: identificazione dei punti di accesso a un sistema e sue vulnerabilità, derivabili tramite analisi attiva (uso di strumenti come *nmap* in maniera più o meno aggressiva) o passiva, leggendo informazioni già da fonti note (come *Shodan.io* partendo da un indirizzo IP): ad esempio, possiamo notare come siano presenti ed esposti servizi con versioni obsolete / vulnerabili

2. **Armamento**: creazione del vero e proprio malware da utilizzare nell'attacco, nell'intento di sfruttare la vulnerabilità individuata - ad esempio creiamo un pacchetto IP con scapy

3. **Consegna**: il payload creato deve essere consegnato alla vittima in qualche modo, come una mail di phishing o un form di una pagina del sito web - quindi inviamo il pacchetto al server al suo indirizzo IP pubblico

4. **Exploit**: nella realtà dei fatti, si sta sfruttando una vulnerabilità individuata - nel nostro caso reale, potremmo aver scatenato una RCE (Remote Code Execution) o altre tecniche

5. **Installazione**: dopo la riuscita dell'exploit, si scarica e avvia il payload malevolo, cercando di bypassare strategie esistenti di rilevazione, usando metodologie come obfuscation - nell'esempio, possiamo installare una reverse shell ad avvio automatico in file come ‘ /.bashrc‘ o ‘ /.profile‘

6. **Command and Control (C&C)**: andiamo a instaurare un canale di comunicazione tra noi e la vittima, così da controllarlo da remoto - qui possono essere utili strumenti come Havoc C2.

7. **Azioni sugli obiettivi**: eseguiamo le azioni più disparate, in base a ciò che vogliamo fare noi come attaccanti, come leggere un file contenenti informazioni sensibili o eseguire il dump del database

1.3 Sistemi di protezione

Ora che abbiamo visto tutte queste minacce, viene spontaneo chiedersi come sia possibile proteggersi.

L'**antivirus** e' lo strumento base per la protezione dei singoli computer. Gli antivirus tipicamente ricercano i malware noti (utilizzando database di firme) analizzando i file memorizzati su disco. Il limite dell'approccio basato su firme (o signature dei malware) è quello di scovare solamente i malware noti.

Ben più potente è l'EDR, acronimo di **Endpoint Detection and Response**, che sfrutta l'*analisi comportamentale* per rilevare e contrastare anche minacce sconosciute, basandosi sui loro comportamenti. Ad esempio, se vediamo che, dopo aver aperto un file scaricato da Internet, il processo del programma che ne permette la visualizzazione inizia a fare operazioni fuori dal normale funzionamento, come aprire una shell o modificare file o registri di sistema, sicuramente andrà terminato e riportato come incidente, che poi andrà gestito da chi di competenza. La sostanziale differenza è il passaggio da semplice analisi su disco a una profonda analisi comportamentale, sfruttando strumenti forniti dal sistema operativo come Microsoft ETW *(Event Tracing for Windows)* o eseguire l'hooking delle syscall manualmente - tecnica ben più complessa ed error-prone.

Infine, con un XDR (**eXtended Detection and Response**) andiamo ad allargare gli elementi sotto la protezione del software in questione, permettendo un'integrazione anche con reti, firewall, log di sicurezza e molto altro. In questo modo, ha la potenza di aggregare e correlare eventi da più fonti, rilevando minacce sofisticate che verrebbero altrimenti ignorate. Un esempio in questa categoria è *SentinelOne XDR*.

1.4 Cyber Threat Intelligence

L'area di Threat Intelligence si occupa di raccogliere tutta una serie di dati dai vari elementi, per poi aggregarli e compiere analisi sulle informazioni ottenute, al fine di rilevare minacce anche sofisticate e prendere decisioni sulla sicurezza in maniera più veloce e consapevole.

La Threat Intelligence è un tassello estremamente importante nella sicurezza di un sistema informatico, soprattutto se unito al MITRE ATT&CK framework: documentando i profili con cui l'avversario opera (come `APT29` - i profili sono insiemi di attività tipiche del modo di operare di alcuni gruppi nel panorama cyber), ma anche individuando la tecnica più efficacemente così da raggruppare tra loro più attacchi con caratteristiche comuni. In questo modo, è possibile focalizzarsi maggiormente sulle tecniche che sono maggiormente usate in un dato profilo di attacco.

Esempi di come particolari avversari, delineati sotto un profilo, usano le proprie tecniche sono documentati nella pagina ATT&CK corrispondente. Studiandone la metodologia di attacco, è possibile replicarlo in fase di emulazione *(Adversary Emulation)* per capire fino in fondo come difendersi da attacchi similari, e testare le proprie difese, nonché eseguire test di rilevazione.

Per questo e altri motivi, come vedremo, nella prima fase di analisi integreremo questo framework per la categorizzazione delle capabilities di un eseguibile.

Un altro importante ruolo della Threat Intelligence, maggiormente vicino al progetto realizzato, è quello di analizzare i malware coinvolti negli attacchi noti o forniti da clienti qualora ne chiedessero l'analisi approfondita. Proprio a questo scopo, il proprio lavoro viene nettamente agevolato da strumenti di analisi automatica più estendibili possibile, così da avere fin da subito tutti i dati su cui lavorare.

1.5 Contesto d'uso del progetto

Una volta vista una panoramica di questo ambito della sicurezza informatica, andiamo ad analizzare il contesto in cui si va ad inserire il progetto realizzato.

L'azienda ospitante offre, come suo core business, sistemi di EDR e XDR. Questi sistemi usano programmi Agent installati nei vari endpoint, che sonde usate per analizzare il traffico di rete nei vari livelli ISO-OSI, nonché analisi sui dispositivi OT.

Soprattutto in riferimento all'Agent installato sugli endpoint, esso andrà a rilevare eseguibili con comportamenti anomali o sconosciuti e li andrà ad inviare al sistema di Threat Intelligence, che valuterà con una data confidence se siano o meno malevoli. La capacità di eseguire detection di alta qualità è uno dei tratti distintivi tra le diverse soluzioni di sicurezza esistenti sul mercato e ha una diretta ripercussione sulla soddisfazione del cliente finale dell'azienda, oltre alla propria reputazione come brand. Proprio in questo punto cruciale, viene l'esigenza dietro il progetto realizzato.

1.5.1 Il problema da risolvere

Lo scopo principale è la costruzione di un sistema proprietario per l'analisi automatizzata di malware, eliminando le ultime dipendenze con servizi terzi quali VirusTotal, sia per un fattore economico che di estensibilità. Infatti inserire un nuovo tool all'interno di VirusTotal è pressoché impossibile per un utente, mentre è ragionevole disponendo di una soluzione in-house, come quella qui realizzata.

Il requisito della modularità ha inoltre condizionato la progettazione, richiedendo anche lo sviluppo di un'interfaccia astratta comune per integrare futuri componenti. Infine devono essere fornite le API per l'integrazione con servizi che ad oggi ancora non esistono o sono in uno stato embrionale, come la Threat Intelligence automatizzata.

1.6 Architettura generale

Il progetto si andrà ad integrare nella nuova architettura di sistema proprietaria dell'azienda ospitante. Seppur è ancora in costruzione, per cui alcuni aspetti potrebbero variare tra il momento della scrittura e la loro effettiva realizzazione, ne viene delineato lo stato dell'arte.

Il punto di partenza è rappresentato dall'Agent, nominato poco fa e installato sui vari endpoint ove possibile, che raccoglie eseguibili sospetti e che reputa necessitino un'analisi più approfondita. Siccome spesso si trova all'interno di una rete isolata, dove non è permessa la comunicazione con qualsiasi host Internet, può inviare gli eseguibili al COS. Più precisamente, inizialmente l'Agent non conosce ancora se siano già presenti analisi, così per questioni di efficienza invia solo il suo hash, inviando poi l'intero file solo in caso di esito negativo.

Il COS si occuperà di mantenere una coda di analisi, per non sovraccaricare il resto dei sistemi. Invierà il file o l'hash ad FSL, che si pone come middleware per le API della Threat Intelligence.

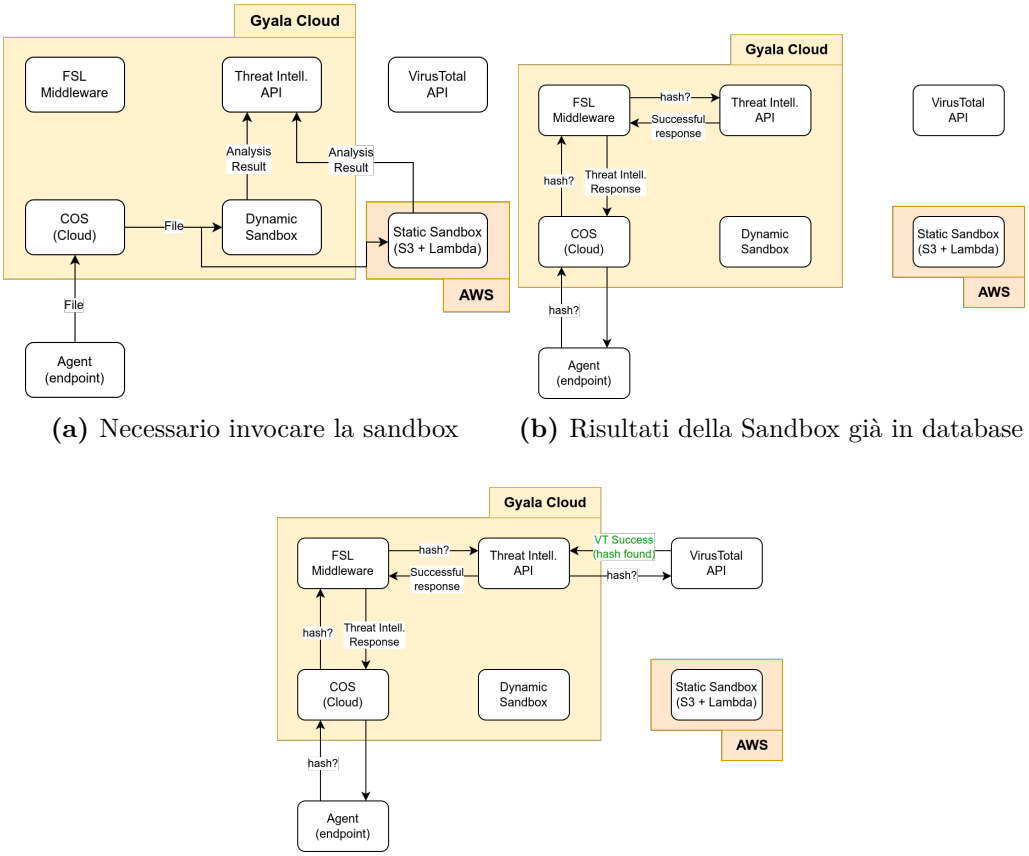
Attualmente viene interrogata l'API di VirusTotal Enterprise per ottenere il maggior numero di informazioni, ma come abbiamo detto questa è la parte più critica a causa degli ingenti costi (fig. 1.3c).

In caso negativo, verrà invocato il progetto qui realizzato. Quindi, l'Agent invierà il file (non più l'hash) al COS, che lo inoltrerà alla nostra Sandbox, che si compone di una parte di analisi statica e di una dinamica (fig. 1.3a). Come vedremo, la fase di analisi statica avrà come trigger l'upload di un file su uno specifico bucket S3.

Il sistema di analisi qui realizzato, chiamato per brevità anche *Sandbox*, si compone di due parti, indipendenti l'una dall'altra al fine di mantenere una massima modularità. Qui sono riportate come introduzione le due architetture in linea di massima, che verranno poi ben dettagliate nelle apposite sezioni di seguito.

Per quanto concerne l'aspetto statico, ci si avvarrà di Amazon Web Services per il funzionamento, sfruttando principalmente i servizi di cloud storage S3 Bucket e di esecuzione Lambda, per minimizzare i costi e avere un sistema flessibile. Una volta analizzato il file, la Threat Intelligence, mediante le sue API, riceverà gli stati di inizio e fine del processo, inclusi i risultati in un formato JSON concordato. Lo andrà a salvare e lo userà nelle future richieste con lo stesso hash (fig. 1.3b).

Al contempo, l'aspetto dinamico si avvale di una già nota sandbox open-source quale Cuckoo Sandbox, su cui è stato costruito il resto del sistema completo, andando a installare il software, configurare il sistema a livello di firewall e SystemD Unit, nonché creare nuovi moduli al fine di integrare tool nuovi, e un sistema di reporting che si basa su un client Python sviluppato appositamente per lo scopo.



(a) Necessario invocare la sandbox

(b) Risultati della Sandbox già in database

(c) Vecchio flusso prima della creazione della Sandbox

Figura 1.3. Schema generale di funzionamento dell'architettura nei vari casi

Capitolo 2

Analisi statica

Il primo passo di questo processo di analisi riguarda in primo luogo un approccio di tipo statico.

Questo tipo di analisi è usato per fare una prima esaminazione di file di vario tipo, senza però eseguirli su un sistema. Per arrivare a tali risultati, si ispeziona il codice del file del malware, alla ricerca di varie caratteristiche, in base a quale strumento stiamo usando e al suo scopo. Possiamo andare alla ricerca, più o meno approfondita, di chiamate di sistema particolari, indizi di offuscamento o crittografia tramite un'analisi dell'entropia di alcune sezioni del file, nonché più direttamente degli IoC già noti. Si possono usare numerose tecniche, tra cui:

- **Disassemblaggio o decompilazione** al fine di andare ad esaminare il codice, più o meno ad alto livello, del sorgente originale dell'eseguibile, anche se spesso questa pratica resta difficoltosa per tutte le ragioni dietro la reverse-engineering, con particolare attenzione degli autori del malware di rendere tecniche come questa il più difficili possibile da applicare;
- **Analisi delle stringhe** per identificare IoC particolari come URL noti, indirizzi IP, o anche semplicemente nomi di chiavi di registro o file con cui normalmente non si dovrebbe interagire in un funzionamento classico;
- **Estrazione dei payload** integrati all'interno del file, come exploit nascosti / crittografati o anche lo stesso codice del vero eseguibile, come nel caso dei file pacchettizzati, che tratteremo nelle successive sezioni

Infine, non necessariamente si deve trattare di un eseguibile: può anche essere analizzato un documento Office o un file PDF, anch'essi noti per essere potenziali vettori di attacco, includendo delle macro o del codice al proprio interno.

2.1 Struttura binari

Prima di andare nel dettaglio di quali tecniche sono state adottate per avere un sistema di analisi statica quanto più completo possibile, bisogna andare a illustrare sommariamente la struttura di un file binario.

Innanzitutto, un file eseguibile contiene sia codice che dati, ed è diviso in **sezioni**, che possono avere diversi permessi di lettura (read-only) o scrittura (read-write).

Questo è un dettaglio più importante di quanto ci aspetteremmo perché ad esempio un eseguibile potrebbe compiere tecniche di modifica della propria sezione di codice, impostandola come read-write, potendo eludere così in maniera piuttosto significativa tecniche di analisi statica, poiché il vero codice del software sarà creato durante la sua stessa esecuzione, e nella fase statica (come dice il nome stesso) non andiamo a lanciare alcun programma.

Un'altra area comune e rilevante è la **symbol table**, o "tabella dei simboli". Questa contiene riferimenti a funzioni o variabili poste in altri file o librerie esterne, che nella prima fase di *Assembling* sono stati trasformati in degli *object file* distinti, e come tali sono rilocabili, ossia non dipendono già da particolari indirizzi in memoria. Quindi contengono riferimenti simbolici ad altri componenti, di cui avranno bisogno nella fase di Linking.

Proprio successivamente al Linking, abbiamo un singolo file binario, e questo avrà una tabella dei simboli più ridotta, ma pur sempre contenente riferimenti a funzioni contenute in librerie di sistema (si parla di librerie linkate dinamicamente). Un attaccante potrebbe aumentare il più possibile il numero di librerie linkate staticamente, così da nasconderne il proprio utilizzo. Un'altra tecnica molto usata in vari contesti è lo *stripping*, eliminando dalla symbol table tutti i riferimenti alle funzioni esportate nel file stesso. Possiamo vedere un esempio di questo fenomeno nel caso dei file ELF su Linux in figura 2.2.

Di seguito, vediamo le principali peculiarità a seconda del sistema operativo a cui ci riferiamo.

2.1.1 ELF

Il formato ELF *(Executable and Linkable Format)* è proprio dei sistemi Linux, ma non solo, infatti è usato anche in vari microcontrollori, o console da gioco come la Nintendo Wii e la PlayStation [4]. Viene usato per file eseguibili, file object comunemente indicati con l'estensione `.o`, shared libraries aventi estensione `.so` o core dumps.

Si compone di un header, descritto completamente nel file `sys/elf.h` e visibile in dettaglio in figura 2.1, contenente varie informazioni tra cui: il tipo di file (`e_type`), il magic number con cui possiamo riconoscere che si tratta di un ELF, l'architettura per la quale è stato compilato (`e_machine`) e l'indirizzo di memoria virtuale, indicato come un offset all'interno del file, che il sistema operativo dovrà eseguire per lasciare il controllo al nuovo processo nato (`e_entry`).

Ma al suo interno troviamo anche le sezioni. Di seguito sono elencate le più rilevanti, con associati i possibili indicatori a cui prestare attenzione: [4]

- `.text` contiene il codice principale del programma, e rappresenta il focus principale di ogni analisi sul file binario; ha il tipo `SHT_PROGBITS` affinché sia eseguibile ma non scrivibile, protezione assolutamente voluta in un programma standard

- `.data` e `.rodata` contengono i dati del programma, come le variabili o le costanti, a seconda che sia una sezione scrivibile o meno - rilevare un valore di entropia molto alta in questa sezione può essere indice di uso di compressione o crittografia, e va valutato caso per caso se è un comportamento che ci

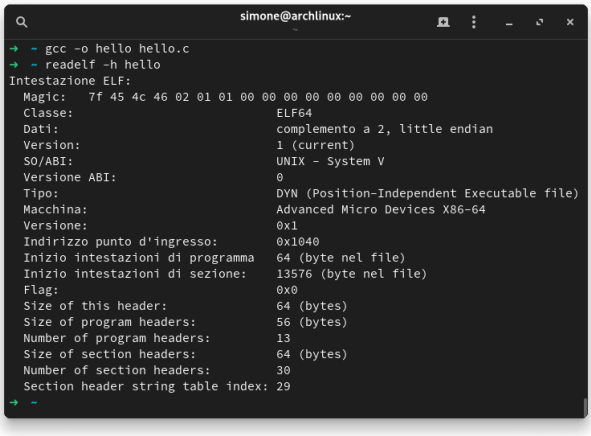
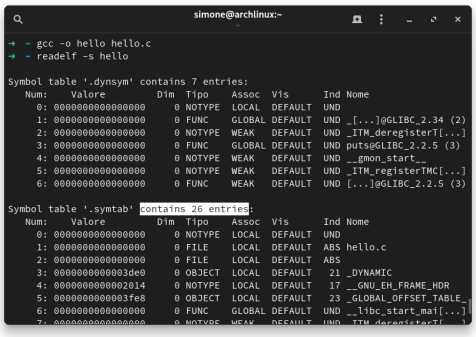


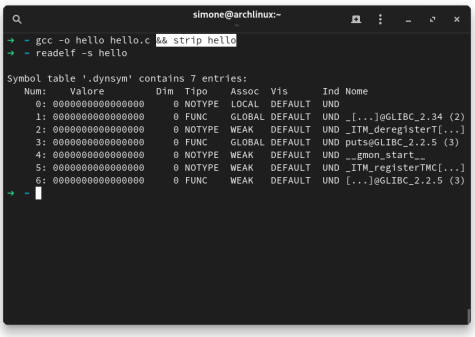
Figura 2.1. Header di un file ELF, letto con readelf

aspetteremmo o figura come indicatore di qualcosa di malevolo, proprio perché manifesta volontà dell'autore di rendere sempre più difficile l'analisi statica. Anche programmi coperti da copyright possono adottare misure di offuscamento, ma non per questo sono da considerarsi malevoli

- `.init` e `.fini` contengono le procedure di inizializzazione e finalizzazione da eseguire prima del main o prima della terminazione del processo a seguito della fine del normale flusso di esecuzione



(a) Symbol table originale



(b) Symbol table dopo strip

Figura 2.2. Confronto delle symbol table prima e dopo lo strip

2.1.2 PE

PE sta per **Portable Executable** ed è il formato usato nei sistemi Microsoft, derivato dal formato *COFF* di Unix, motivo per cui è spesso identificato come PE/COFF. [4]

Viene usato per vari tipi di file, tra cui eseguibili, DLL (Dynamic Link Libraries) e object file.

Anch'esso ha il proprio magic number ai fini di essere identificato correttamente, ed è composto da più header, dove possiamo notare tutta la sua evoluzione storica come formato. Inizialmente troviamo header specifici di MS-DOS, e successivamente altri più caratteristici delle nuove versioni dei PE, includendo sia informazioni di base comuni agli ELF, come il numero di sezioni, di simboli e l'architettura, ma anche la firma digitale dell'immagine del programma per proteggersi da tampering e verificare che l'eseguibile sia realmente proveniente dalla corretta azienda, a meno di leak delle chiavi privati usate per la firma stessa. [1] Quest'ultima caratteristica può semplificare il lavoro di un analista, comunicando che si tratta di un file originale rilasciato da esattamente una specifica azienda nota. Non deve tuttavia essere una garanzia al 100%, vista la possibilità di leak delle chiavi di firma, seppur remota. In alcuni ambienti con politica zero-trust, può essere ritenuto idoneo vietare l'esecuzione di file non firmati o di produttori inaspettati.

2.2 Capa: capabilities

Per i test che vedremo in questa sezione e nelle successive, sono stati scaricati **40 sample** di malware reali da VirusTotal casualmente, e nelle varie fasi di miglioramento ed evoluzione, soprattutto in merito all'analisi delle capabilities, sono stati fatte numerose prove usando come campioni proprio questi sample.

Per *capabilities* intendiamo le funzionalità che va a usare il programma, ossia ciò che è capace di svolgere durante la propria esecuzione. Esempi di capabilities possono essere la connessione a un server remoto, la modifica di chiavi di registro, o la creazione di un processo figlio, solo per citarne qualcuna.

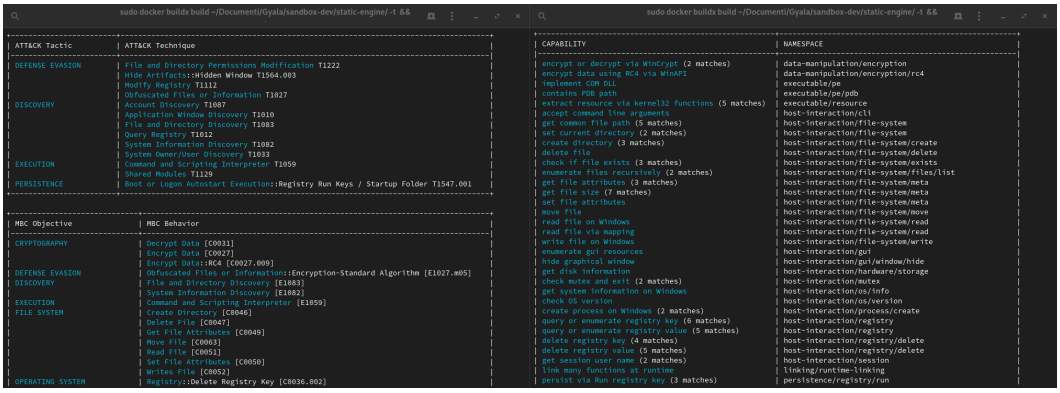


Figura 2.3. Esempio di esecuzione di capa su pafish64

Questa estrazione infatti sfrutta lo strumento **capa**, [2] visto in figura 2.3, tool open-source creato da Mandiant, una delle aziende leader del settore e ora parte di Google, che permette di andare ad individuare delle capabilities all'interno di un file eseguibile sfruttando delle regole aventi un proprio formato. Funziona andando prima ad estrarre delle *features* come le stringhe incluse, codice disassemblato, control flow

[1] Maggiori dettagli sono disponibili alla pagina `https://learn.microsoft.com/en-us/windows/win32/debug/pe-format`

[2] `https://github.com/mandiant/capa`

o file importati. Poi vengono lanciate le regole su queste features al fine di trarne una conclusione logica e ottenere informazioni valide a fronte di un match con una o più regole capa.

Si è rilevata la *scelta ottimale* per il progetto per la sua natura open-source, per la grande azienda che è dietro la sua realizzazione, ma soprattutto per la potenza espressiva delle sue regole. Possono essere scritte in un file di testo in formato YAML, seguendo la semplice struttura richiesta da capa, ed essere in futuro aggiunte nello strumento realizzato, sia preparate internamente che estratte da altri contributori in repository pubblici. Infine, è possibile anche organizzarle per tag (chiamati anche *namespace*), così decidendo volta per volta quali specifiche regole eseguire, garantendo massima flessibilità.

Il processo di feature extraction si basa in gran parte sull'header che abbiamo appena menzionato nella sezione 2.1, comprese informazioni sulle funzioni esportate, tabella dei simboli (SymTab) e sezioni. Inoltre, il processo di disassemblaggio è molto complesso perché punta sia ad individuare features quali API calls, istruzioni speciali o riferimenti di stringhe/numeri poste in altre sezioni dell'eseguibile (come `.rodata` per le costanti), ma anche a ricostruire il control-flow minimizzando falsi positivi in caso di dead code o per distinguere tra funzioni e altri scope. Alla base di questo step, c'è `ViviSect Feature Extractor`. [2]

Una volta ottenute tutte le features di interesse, si procede a fare il match con le regole. Una regola capa è una combinazione logica strutturata di match di varie features più di basso livello, per giungere alla conclusione di presenza o assenza di una determinata capability all'interno del software in analisi, espresse sotto forma di file `YAML`.

Nelle regole, può venir inoltre specificata la tecnica e quindi la tattica, secondo il MITRE ATT&CK Framework (sez. 1.1.1), come visibile nell'output del comando (fig. 2.3). Questa rappresenta un grande vantaggio per l'integrazione con altri strumenti di Threat Intelligence, per le stesse ragioni favorevoli dell'adozione del framework, già discusse.

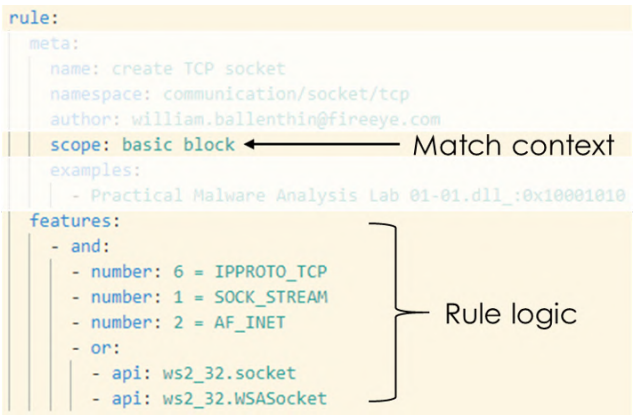


Figura 2.4. Regola capa per rilevare la creazione di un socket TCP

2.2.1 Parsing dell'output di capa

Capa, al contrario di ciò che vedremo di seguito, è un tool da usare da CLI e offre un output testuale, come in figura 2.3.

Siccome al nostro strumento serve un output più machine-readable da restituire come risultato del processo di analisi, è stato scritto in Python un **parser** in grado di trasformare quel testo in un JSON. Le chiavi principali sono come le 3 tabelle, ossia `capabilities`, `tactics` e `objectives`, oltre alla prima parte di `info` presente all'inizio, seguendo la struttura già fornita dall'output testuale.

Analizziamo com'è stato eseguito il parsing a un livello più astratto. Prima di tutto, notiamo dalla figura 2.3 come le linee vuote dividano le sezioni, quindi possiamo sfruttare questo fatto per stabilire la fine dell'una e il conseguente inizio dell'altra. Poi, le linee che ci interessano iniziano per `|`, escluso l'inizio per `|-`. Così facendo arriviamo a un output similare a quanto in figura 2.5.

Notiamo come la prima linea di ogni sezione contenga il nome della sezione stessa, fatta eccezione per le informazioni che però sono sempre la prima informazione contenuta, così da stabilire quale sotto-parser usare specifico per quella porzione di testo.

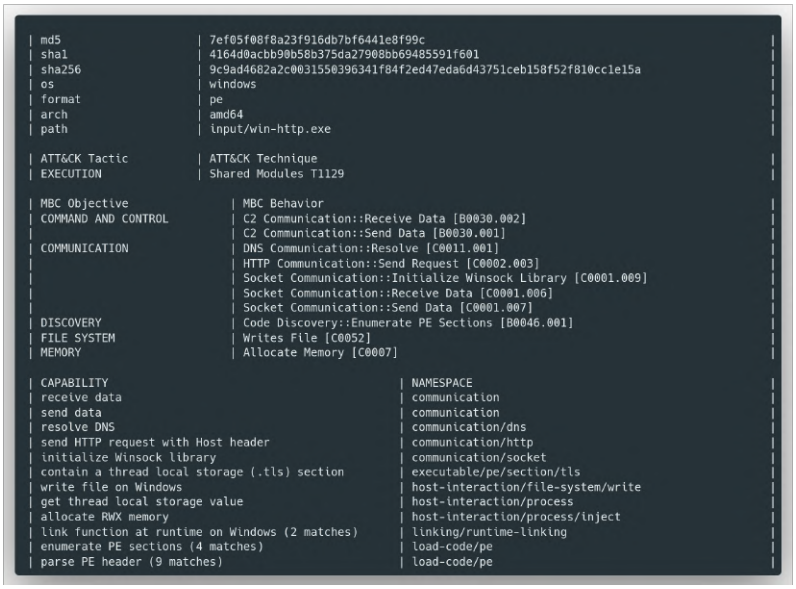


Figura 2.5. Prima trasformazione dell'output di capa

Possiamo stabilire una tecnica per ogni sezione, come segue:

- **Info** potrà essere vista come un dizionario (chiave-valore) con come chiave il testo della prima colonna, e il valore preso dalla seconda colonna alla riga corrispondente;
- **Tactics** e **Objectives** possono essere viste entrambe come dizionari multi-valore, dove il valore è una lista - infatti, se una riga ha una chiave (prima colonna non vuota), creeremo una nuova lista di valori, altrimenti potrà essere appeso alla lista corrispondente alla chiave precedente

- **Capabilities** segue una logica analoga ma invertendo le colonne: qui la chiave è il namespace e come valore c'è la lista di capabilities stesse, con un maggiore parsing, che include sia il nome della capabilities che il numero di match (in figura 2.5 notiamo questa situazione nelle ultime 3 righe dell'output)

In questo modo, si arriva ad un formato JSON machine-readable, che astrae dallo specifico formato di output di capa, che potrebbe cambiare tra major release, e allo stesso tempo organizzato in maniera sufficientemenete logica per essere consultato dall'analista di sicurezza nell'ambito del proprio studio del sample.


```
{
  "info": {
    "os": "windows",
    "format": "pe",
    "sha256": "9c9ad4682a2c0031550396341f84f2ed47eda6d43751ceb158f52f810cc1e15a"
  },
  "tactics": [{
    "tactic": "EXECUTION",
    "techniques": [ "Shared Modules T1129" ]
  }],
  "objectives": [{
    "objective": "COMMAND AND CONTROL",
    "behaviors": [
      "C2 Communication::Receive Data [B0030.002]",
      "C2 Communication::Send Data [B0030.001]"
    ]
  }],
  "capabilities": [{
    "namespace": "load-code/pe",
    "capabilities": [
      {
        "capability": "enumerate PE sections",
        "matches_count": 4
      },
      {
        "capability": "parse PE header",
        "matches_count": 9
      }
    ]
  }]
}
```


Listing 1. Esempio sintetico del formato di output successivo al parsing

2.2.2 Analisi dell'uso delle risorse

Vogliamo analizzare l'uso di risorse da parte di capa con una varietà di file diversi, ossia i 40 sample precedentemente citati da VirusTotal, in termini di spazio e tempo,

provando a cercare delle correlazioni con altri dati a nostra disposizione, al fine di eventualmente poter eseguire delle ottimizzazioni. Bisogna ricordare infatti che il ben più ampio sistema con cui questo progetto si andrà in futuro a interfacciare lavora diverse migliaia di sample ogni mese, per cui è fondamentale ridurre gli sprechi.

All'interno di una macchina virtuale, è stato installato il tool `capa` insieme ai sample da mettere sotto esame. Prima di fare ogni tipo di studio, abbiamo bisogno di estrarre dei dati, ottenibili con uno script Python che legge ed esegue un basilare parsing dei log del comando, tramite il suo standard error, nonché il suo tempo di esecuzione.

Uno dei dati più significativi è il numero di funzioni o blocchi che devono essere analizzate dal feature extractor. Usando matplotlib, andiamo a validare questa supposizione o confutarla, visualizzando i due elementi in un grafico (fig. 2.6).

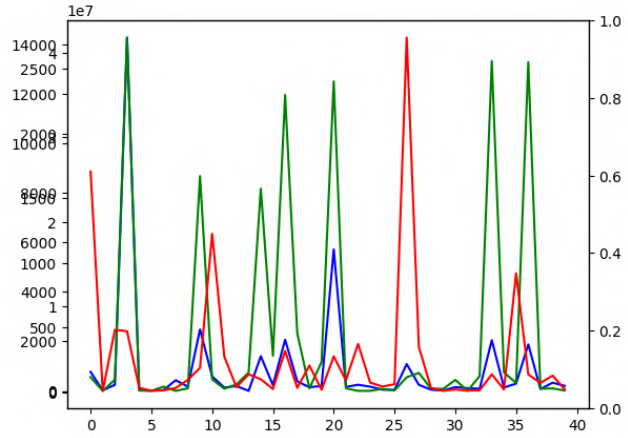


Figura 2.6. Correlazione tra tempo di esecuzione (blu) e numero di funzioni/blocchi (verde), assenza di relazione con la dimensione del file (rosso)

Si evince come ci sia correlazione tra il numero di funzioni e il tempo di esecuzione dello strumento, mentre sia pressoché assente tra altre variabili rappresentate in figura. Siccome su questa prima metrica non possiamo intervenire in alcun modo, ci si è focalizzati sull'evitare a monte l'analisi con capa di file che non sono supportati. Infatti, si è notato come non tutti gli eseguibili sono supportati da capa, proprio per i motivi precedentemente elencati riguardanti i file offuscati o pacchettizzati. A questo scopo sono presenti, all'interno del set standard di capa, delle regole (sotto il namespace `internal/limitation/file` [3]) che se presentano un match col file in analisi, capa si arresterà non ritenendo più affidabili i propri risultati.

2.2.3 File non supportati

Per prima cosa è stato notato come il tool restituisse un diverso *exit code* in base alla sua corretta esecuzione o meno (listing 2). Da ciò, possiamo rilevare quando un file poteva essere scartato a monte o l'esecuzione di capa ha portato a risultati utili per cui le risorse sono state ben spese.

[3] `https://github.com/mandiant/capa-rules/tree/master/internal/limitation/file`

```
function test_file_with_capa() {
    capa "$1" 2>&1 | grep "WARNING:capa: Identified via"
    echo "Exit code (capa): ${PIPESTATUS[0]}"
}

$ test_file_with_capa ./executables/visual_basic_file.exe
WARNING:capa: Identified via rule: (internal) Visual Basic file limitation
Exit code (capa): 14

$ test_file_with_capa ./executables/standard_pe.exe
Exit code (capa): 0
```

Listing 2. Distinzione tra gli exit code di capa

Da qui, abbiamo potuto categorizzare i vari sample in gruppi sulla base del tipo di file, che sia standard (`pe`) o non supportato da capa (`packed`, `installer`, ...). Creiamo quindi uno script Python che sia in grado di automatizzare questa azione, sia sulla base dell'exit code che dell'output sullo stderr. Otteniamo un JSON come il seguente (estratto per sintesi):

```
{
  "./b9c32debb4.exe": "installer",
  "./9f79ea539a.exe": "pe",
  "./4726c42b25.exe": "visualbasic",
  "./e49c42d0b1.exe": "pe",
  .....
}
```

La distribuzione della tipologia dei file è riassumibile più graficamente nella figura 2.7, dove notiamo come circa la metà dei sample è difficilmente analizzabile con capa per difetti congeniti dell'analisi statica. Siccome non possiamo agire direttamente su questo aspetto per eseguire il tool, il prossimo passo sta nel quantificare lo spreco di risorse che si verifica analizzando le capabilities di file che poi porteranno a un fallimento del processo, e se può essere conveniente a livello di trade-off costruire una soluzione per salvaguardare l'uso di risorse saltando del tutto la sua esecuzione (codice 3).

```
times = {
    k: v['elapsed_time']
    for k, v in times.items() if file_types[k] != 'pe'
}
avg_wasted_time = sum(times.values()) / len(times)
print(f'{avg_wasted_time:.2f} sec. per unsupported file (avg)')
# 58.77 sec. per unsupported file (avg)
```

Listing 3. Calcolo del tempo sprecato eseguendo capa su file non supportati

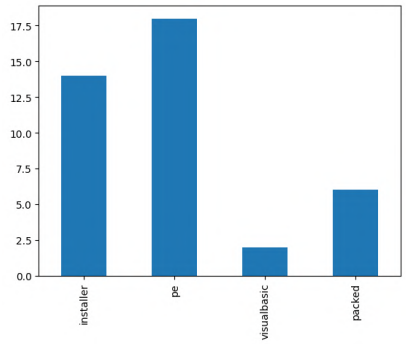


Figura 2.7. Divisione dei sample per tipologia

Notiamo che impiega **un'eccessiva quantità di tempo** per eseguire il match delle regole le quali indicano a capa l'impossibilità a proseguire con una soddisfacente confidence, con una media di quasi *un minuto per file*, che moltiplicato per centinaia di migliaia di file diventa una grande quantità di tempo totale che potrebbe essere usato in maniera migliore.

Ci basta avere i dati di ciò che stiamo trattando, ad esempio l'output desiderato in questo caso sarà del tipo:

```
{
    "capa": {
        "format": "packed",
        "arch": "i386",
        "os": "windows"
    }
}
```

Come conclusione, si è ritenuto molto valido realizzare un sistema per riconoscere preventivamente se si tratti di un file che non è possibile analizzare con capa.

2.3 Riconoscitore custom del tipo di file

Partendo dalla necessità di velocizzare l'operazione di riconoscimento del file, in sostituzione dell'esecuzione del comando `capa -r anti-analysis` che può portare a richiedere diversi minuti a seconda del numero di funzioni presenti nel file, come visto in figura 2.6, dobbiamo ricorrere a una diversa soluzione.

Dato che si tratta di un problema di enumerazione, non aveva senso andare a creare e mantenere un proprio riconoscitore di ogni possibile formato di un file. Ad esempio, già solo per quanto riguarda i file pacchettizzati, esistono tantissimi packer e tanti ne esisteranno in futuro.

Inoltre, spesso malware usano packer personalizzati, che hanno leggere differenze da quelli noti e si correrebbe il rischio di creare uno strumento poco efficace. Per questi e altri motivi quali il tempo a disposizione, si era inizialmente scelto di usare come base uno strumento noto (`Detect-It-Easy`) per fare una prima analisi, affiancato da uno script Python custom che decide quali operazioni svolgere a

seconda del tipo di file, va a fare il parsing del JSON dato in output e si integra nello script Bash creando un layer di astrazione col resto del workflow. Infatti, nel caso servisse modificare lo strumento sottostante, sarebbe sufficiente adattare la traduzione dall'output specifico del tool al contratto stabilito col resto del programma per rendere seamless questa variazione, anche sostanziale. Tuttavia, ci sono alcuni tipi che anche Detect-It-Easy fallisce a riconoscere, portando poi a un crash di capa, dopo aver già consumato risorse per la propria esecuzione parziale.

Per questo motivo è stato creato un **custom detector**, scritto in Python, che implementa in maniera molto più efficiente e diretta le sole regole di capa che riconoscono i file specifici su cui non è in grado di lavorare.

Di seguito possiamo osservare come funziona per dei tipi famosi di questi strumenti, ovvero il packer UPX e l'installer InnoSetup. Sono state usate caratteristiche e regole note [4] per eseguire la detection.

La lettura degli header del file PE sfrutta la libreria Python `pefile` per non eseguire l'intero parsing a mano dei byte degli header e non essere dipendenti da implementazioni native dell'OS come con l'uso dell'header `<windows.h>` in un programma C.

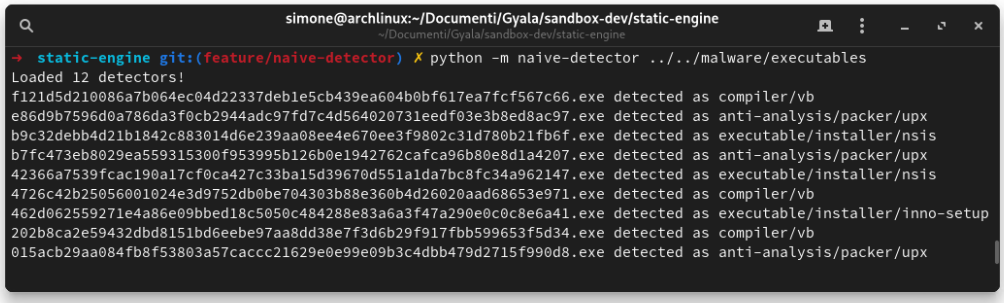


Figura 2.8. Custom file detector output

2.3.1 UPX

Le regole più comuni [5] per riconoscere UPX impongono di leggere le sezioni, e vedere se queste contengono `UPX0` o `UPX1`.

Facciamo una comparazione tra la regola capa per il rilevamento e il codice del riconoscitore.

```yaml
features:
    - format: pe
    - or:
      - section: UPX0
      - section: UPX1
```

```python
def is_upx(file_path):
    file_sections = get_section_names(file_path)
    upx_sections = [ "UPX0", "UPX1" ]
    for upx_section in upx_sections:
        if upx_section in file_sections:
            return True
    return False
```

[4] https://github.com/mandiant/capa-rules/tree/e88db21de4d4cf9f7abec9177fab11240075036b/anti-analysis/packer

[5] https://github.com/mandiant/capa-rules/blob/e88db21de4d4cf9f7abec9177fab11240075036b/anti-analysis/packer/upx/packed-with-upx.yml

Andando però a sfruttare la libreria Python per il parsing del file PE senza ricorrere a implementazioni native Windows, per facilitare la realizzazione e permettere così l'esecuzione del riconoscitore anche su altri sistemi operativi:

```
import pefile

def get_section_names(file_path):
    pe = pefile.PE(file_path)
    sections_set = {
        section.Name.decode().strip().strip('\x00')
        for section in pe.sections
    }
    pe.close()
    return sections_set
```

Listing 4. Script Python per estrarre i nomi delle sezioni di un PE con pefile

Analisi preliminare con readelf

Si evince come questa tecnica funzioni solo su file analizzati in ambito Windows, mentre su sistemi GNU/Linux osserviamo con strumenti quali `readelf` che un eseguibile pacchettizzato con UPX sia privo di header delle sezioni (in accordo con la regola CAPA stessa, che infatti include questo match solo se ci troviamo su un file PE, come indicato dall'apposita riga `- format: pe`).

Negli eseguibili in formato ELF infatti è ammissibile trovare una situazione come questa, dove l'header delle sezioni è mancante. Infatti, tale informazione non è usata in alcun modo dal kernel durante il caricamento in memoria del programma, nella creazione della process image. Inoltre, non è indicatore di azioni malevoli, ma sicuramente della volontà di rendere l'analisi più difficile, il cui caso può riguardare anche release di software proprietario closed-source.

Analisi dell'entropia

Una caratteristica comune di binari compressi o con payload criptati è l'aumento di entropia. Infatti, sia la compressione che la crittografia vanno ad aumentare questa variabile all'interno di uno stesso blocco di file. Segue la stessa logica il motivo per cui, in generale, si esegue la compressione di un file prima della sua crittazione, altrimenti l'algoritmo di compressione sarebbe molto meno efficace per via della maggiore entropia ottenuta.

Come esempio, è stato preso un eseguibile il cui sorgente è ininfluente, compilato con il compilatore C `gcc` e poi calcolata l'entropia sia prima che dopo averlo pacchettizzato con UPX. I grafici in figura 2.10 sono stati generati tramite l'ausilio di uno script Python creato ad-hoc e sfruttando il calcolo dell'entropia di Shanon.

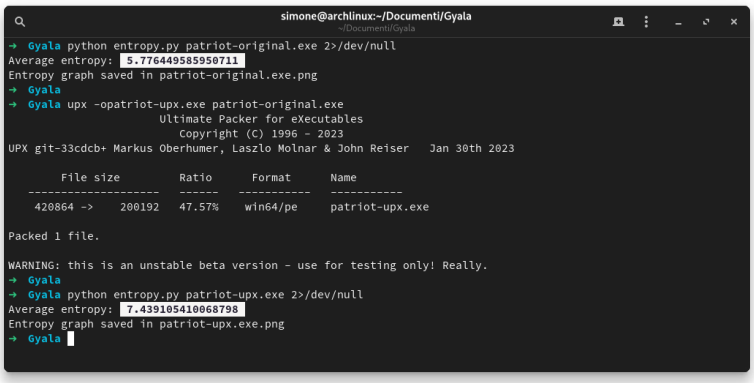
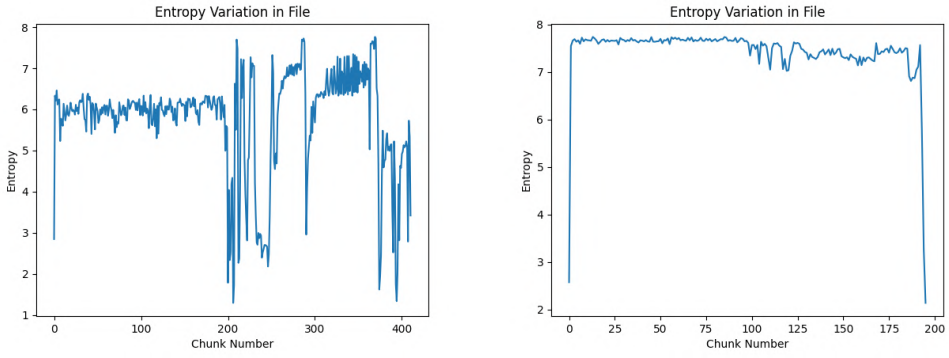


Figura 2.9. Esecuzione dello script per il calcolo dell'entropia e uso di UPX



(a) Entropia del file originale **(b)** Entropia del file pacchettizzato con UPX

Figura 2.10. Confronto dell'entropia prima e dopo l'uso di UPX

2.3.2 Inno-Setup Installer

Con Inno-Setup, possiamo notare come sia ancora più semplice il riconoscimento. Infatti, non abbiamo bisogno di andare a vedere le sezioni, ma è sufficiente controllare la presenza di alcune stringhe nel file.

InnoSetup è un famosissimo installer per Windows e, come tale, non permette un'analisi statica efficace. Infatti, nulla vieta che appaia come lecito per poi andare a runtime a scaricare e installare il vero payload malevolo, ma tale comportamento è affidato alla parte dinamica dell'analisi, dove sarà possibile osservare anche tutti i file scaricati, oltre all'attività di rete e tutte le syscall richiamate (*behavioural analysis*).

Nuovamente, andiamo a paragonare la regola capa alla nostra implementazione:

$$H(x) = -\sum_{i=1}^{n} \frac{count_i}{N} \log\left(\frac{count_i}{N}\right)$$

```python
def calculate_entropy(data):
    entropy, size = 0, len(data)
    if size <= 1: return entropy

    byte_count = [0] * 256
    for byte in data:
        byte_count[byte] += 1

    for count in byte_count:
        if count > 0:
            probability = float(count) / size
            entropy -= probability \
                * math.log(probability, 2)

    return entropy
```

Figura 2.11. Formula dell'entropia di Shannon e codice Python per il calcolo su un chunk

```yaml
features:
    - and:
      - string: /^Inno Setup Setup Data \(/
      - string: /^Inno Setup Messages \(/
```

```python
def is_innosetup(file_path):
    BUFFER_SIZE = 65536
    with open(file_path, 'br') as f:
        match_setup, match_messages = False, False
        buffer = f.read(BUFFER_SIZE)
        overlap = len(b'Inno Setup Setup Data (')

        while len(buffer) > overlap and (not match_setup or not match_messages):
            # Match inside current buffer
            if not match_setup and b'Inno Setup Setup Data (' in buffer:
                match_setup = True
            if not match_messages and b'Inno Setup Messages (' in buffer:
                match_messages = True

            # Read next buffer, allowing for overlap
            buffer = buffer[-overlap:] + f.read(BUFFER_SIZE)

    return match_setup and match_messages
```

2.3.3 Design scalabile e modulare

Si nota immediatamente come una tale implementazione è poco scalabile e presenta numerose ridondanze da rimuovere.

Studiando le regole di interesse all'interno del repository GitHub *capa-rules*, osserviamo come siano principalmente di due categorie: basate su match di stringhe

o basate su match di nomi di sezioni nel file eseguibile.

Da questa osservazione, elaboriamo la struttura del programma software di detection. Precisamente, decidiamo di strutturarlo in classi dove ogni regola è una classe a sé, e individuiamo le seguenti classi padre (figura 2.12):

- `BaseRule` conterrà le logiche di base che possiede una regola, tra cui un getter per il nome e un metodo `match()`

- `StringRule` conterrà il codice unico per tutte le regole che basano il loro match sulla presenza di determinate stringhe all'interno del file, quindi la classe figlia dovrà solo fornire le stringhe da cercare e la logica di match (ad esempio: tutte le stringhe richieste devono essere state trovate = and logico; una sola stringa trovata è sufficiente = or logico)

- `SectionRule` analogamente a prima conterrà le procedure di estrazione delle sezioni dal file, con la logica di match nelle classi figlie, ossia le effettive regole.

Sempre al fine di ottimizzare i tempi di esecuzione di questo rilevamento statico sul file eseguibile, andiamo a creare una classe wrapper sopra `pathlib.Path` che esporrà i metodi per leggere le sezioni, procedura lenta se ripetuta per ogni singola regola, e alcune altre funzioni di utilità come la rilevazione del tipo basilare di file (PE o ELF) eseguibile tramite l'ausilio di `libmagic`, la stessa che permette al comando `file` di operare. Quindi da un lato è stata aumentata l'astrazione tra la business logic del programma e i dettagli implementativi delle librerie sottostanti, dall'altro possiamo aggiungere meccanismi di lazy-loading e caching di tali informazioni richieste.

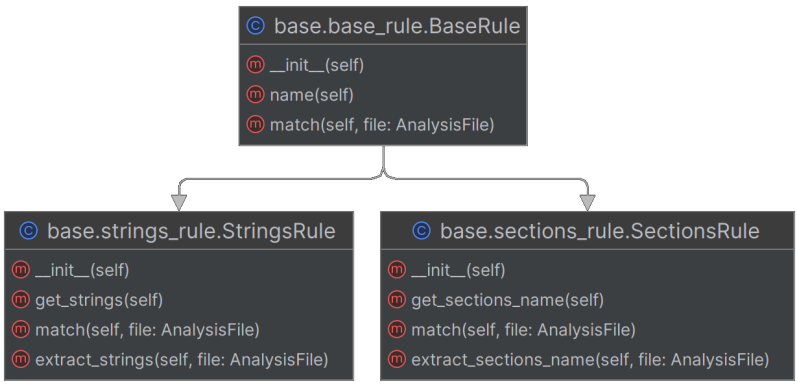


Figura 2.12. Diagramma UML delle classi

```python
class InnoSetupDetector(StringsRule):
    def name(self) -> str:
        return 'executable/installer/inno-setup'

    def get_strings(self) -> list[bytes]:
        return [
            b'Inno Setup Setup Data (',
            b'Inno Setup Messages (',
        ]

    def match(self, file: AnalysisFile) -> bool:
        # && logic
        return len(self.extract_strings(file)) == len(self.strings)
```

Listing 5. Regola di riconoscimento di InnoSetup, usando la nuova architettura

2.4 Yara: signature-based

Un altro strumento adoperato per una più ampia analisi statica del file è Yara. Questo tool utilizza le cosiddette *Yara rule* per descrivere una regola ed eseguire così signature matching.

```
rule ExampleRule
{
    strings:
        $my_text_string = "text here"
        $my_hex_string = { E2 34 A1 C8 23 FB }

    condition:
        $my_text_string or $my_hex_string
}
```

Listing 6. Regola di Yara di esempio

Nella regola di esempio (listing 6) possiamo osservare nel blocco *strings* le stringhe, o la regex, che di cui vogliamo fare il match, e successivamente la condizione per dare esito positivo alla regola.

Questo strumento è ampiamente usato nell'ambito della malware analysis sia su file staticamente che durante la loro esecuzione per fare matching sulla memoria dei processi. Tuttavia, nello scopo del progetto viene relegato all'uso su base statica.

2.4.1 Collezione delle regole

Al contrario di capa (visto nella sez. 2.2), questo tool non viene fornito di un set di regole predefinite, ma dobbiamo andare a collezionarle autonomamente. Tuttavia, essendo molto famoso, si trovano facilmente repository di professionisti e aziende di rilievo che scelgono di rendere le loro regole disponibili alla community. Bisogna

però notare come non tutte le regole siano di qualità e potrebbero in alcuni casi essere controproducenti, generando una quantità eccessiva di falsi positivi o non essere aggiornate o sufficientemente generalizzate da creare falsi negativi. A questo scopo, viene usato lo strumento di *Retrohunt* di VirusTotal Enterprise, che permette di sapere anticipatamente quanti file fanno il match con una data regola, andando a usare l'ampissimo database di VirusTotal. In questo modo, se vedremo un numero troppo elevato di match, probabilmente la regola genera falsi positivi, con un discorso speculare in casi di troppi pochi match.

I principali repository di regole Yara sono di Florian Roth [6] nonché di Apple per quanto concerne MacOS e altre fonti autorevoli in materia.

Per ridurre l'esecuzione di regole inutili, viene sfruttato il rilevamento del sistema operativo precedentemente messo in opera per andare ad eseguire solo regole che abbiano quell'OS come target. Allo scopo, le regole sono mantenute in locale e nell'analizzatore divise in cartelle:

- *windows, macos, linux* a seconda del sistema operativo target
- *common* se sono state ritenute di uso generale per cui ha senso eseguirle in ogni caso, qualunque sia il tipo di file sotto analisi
- *unclassified* conterrà regole che ancora non sono state catalogate da parte dell'analista, ma si ha la necessità di integrarle al momento nel tool, eseguendole sempre; differisce da "common" per il fatto che queste regole potrebbero riguardare anche un solo OS ma temporaneamente le eseguiamo sempre - non possono essere inserite temporaneamente in "common" per una questione di ordine

2.5 Fuzzy hashing

In un'ottica di uso su larga scale, è coerente con le esigenze aziendali avere un sistema di comparazione dei sample. Uno dei modi più semplici ma robusti per eseguire tale confronto è il fuzzy hashing. Questa tecnica consiste nel creare un hash del file che, al contrario di quanto accade nelle funzioni di hashing tradizionali come SHA-256 dove anche una minima modifica al file va a generare un hash completamente diverso, qui la differenza nell'impronta hash è commisurata alle modifiche in un dato file. Ne consegue che file simili, dove è cambiato poco, come una costante o un IoC qualsiasi, avranno un fuzzy hash molto simile.

Una delle funzioni più famose è **SSDeep**. In figura 2.13 è stato creato un file random e una sua versione modificata alterando i primi 8 Byte: possiamo notare come solo il primo carattere sia modificato, di fatto riflettendo una modifica solo nella prima porzione di file. Al contrario, notiamo come un hashing come SHA256 presenta numerosissime modifiche, essendo indicato per tutt'altro scopo.

In questo modo, possiamo lanciare una funzione di distanza tra stringhe e calcolando il numero di caratteri dell'hash diversi, stabilire un grado di somiglianza con altri sample noti. Nel caso di un malware mai visto prima, potremo in realtà

[6] `https://github.com/Neo23x0/signature-base`

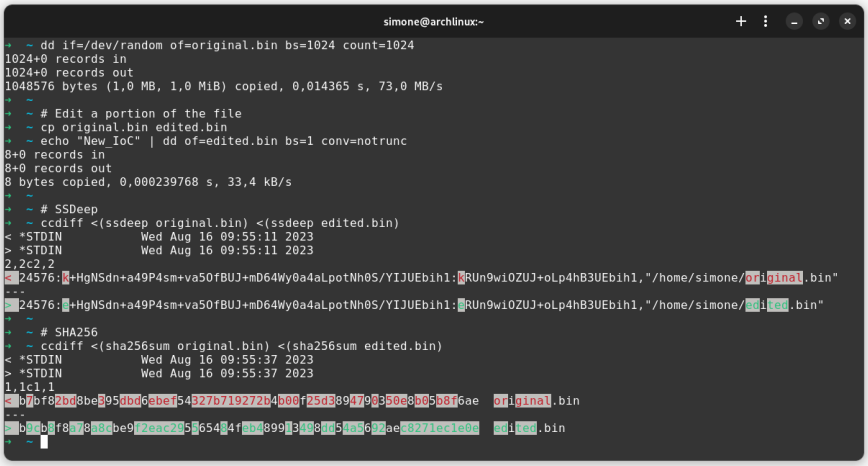


Figura 2.13. Esempio di fuzzy hashing tra file con solo alcuni byte modificati

scoprire che si tratti solamente di una versione modificata di un sample già nel nostro archivio, facendo risparmiare una quantità di lavoro e accellerando i processi.

Il suo funzionamento è intuibile dal formato dell'impronta ottenuta. Si tratta di un *Context-Triggered Piecewise Hash* ed è prodotto andando ad eseguire una funzione hash su ogni porzione di file di dimensione fissata, indicata nella prima parte. Seguono hash per ogni porzione, permettendo di capire chiaramente dove è situata la modifica. Inoltre, è relativamente resistente anche a zero padding, in cui l'attaccante potrebbe pensare di inserire nel payload un numero importante di zero per cercare di rendere più differenti posssibili i due sample.

2.6 Strumenti minori

A dimostrazione della flessibilità dello strumento realizzato, sono stati integrati altri due strumenti importanti ma di minor rilievo rispetto a quelli appena enunciati.

- **Detect-It-Easy**: permette di identificare il tipo di un file eseguendo un'analisi più profonda di quanto faccia `libmagic`, utilizzabile dall'utente attraverso il comando `file` su sistemi Unix, includendo tutta una serie di metadati come il tipo di installer presente, il linker usato, o altre informazioni di questo tipo
- **ExifTool**: un tool Perl in grado di estrarre un gran numero di metadati in una varietà di formati di file, oltre agli eseguibili, incluse foto, PDF, file Office, e così via

```
{
    "diec": {
        "arch": "AMD64",
        "detects": [{
```

```
            "name": "unknown",
            "options": "Console64,console",
            "string": "linker: unknown(2.38)[Console64,console]",
            "type": "linker",
            "version": "2.38"
        }],
        "endianess": "LE",
        "filetype": "PE64",
        "mode": "64",
        "type": "Console"
    },
    "exiftool": {
        "ExifToolVersion": 12.62,
        "FileType": "Win64 EXE",
        "MachineType": "AMD AMD64",
        "ImageFileCharacteristics": "Executable, No line numbers, No symbols, ...",
        "CodeSize": 47616,
        "InitializedDataSize": 122368,
        "UninitializedDataSize": 5120,
        "EntryPoint": "0x14d0",
        "SubsystemVersion": 5.2,
        "Subsystem": "Windows command line",
        "ProductVersionNumber": "0.6.0.0",
        "ObjectFileType": "Executable application",
        "LanguageCode": "English (U.S.)",
        "CharacterSet": "Windows, Latin1",
        "FileDescription": "Paranoid Fish is paranoid",
        "ProductName": "Paranoid Fish"
    }
}
```


Listing 7. Output parziale dei dati ottenuti dai due strumenti minori

2.7 Containerizzazione

Al fine di utilizzare tutti questi strumenti in una maniera organizzata e flessibile, si arriva alla necessità di avere un sistema unico e portatile tra vari sistemi per l'esecuzione. La soluzione migliore a tale scopo è usare un Container Docker. Questo permette di avere un certo livello di isolamento tra lo strumento di analisi e il sistema host sul quale è realmente eseguito, senza tutti gli overhead che comporta una macchina virtuale. Inoltre, possiamo avviarlo a piacimento fornendo direttamente un input da analizzare, montando una cartella come volume read-only, e far si che si elimini automaticamente dopo la sua esecuzione. Dato lo scopo e l'ambito di applicazione del progetto, usare un container permetterà di eseguire il tutto in maniera più semplice possibile su Amazon Web Services (di seguito, AWS).

2.7.1 Flusso di esecuzione

Viene così costruito un chiaro flusso di esecuzione dei vari strumenti, parser e componenti accessorie realizzate, inclusa una chiara gestione degli errori e loro propagazione all'esterno in maniera documentata, al fine di avere una chiara integrazione con gli altri strumenti, quali FSL menzionato nell'introduzione al capitolo 1.6. L'intero flusso è implementato attraverso uno **script sh** che controlla le condizioni dei vari tool in caso debbano essere gestite, o termina l'esecuzione propagando l'exit code allo script richiamante attraverso il flag `set -e` posto all'inizio. Viene usato `/bin/sh` come interprete anziché `/bin/bash` per ridurre la superficie di attacco in caso di future vulnerabilità in qualche strumento, o evitare di introdurne noi alcune date dalla grande varietà di funzionalità offerte da bash e di cui non abbbiamo alcun bisogno.

Per prima cosa, si è deciso di eseguire Detect-It-Easy per avere una prima idea, seppur non comprensiva di tutti i casi limite di capa, del tipo di file che stiamo per analizzare. Il suo output viene salvato in un file JSON associato, che verrà consegnato al termine dell'analisi, ma da cui vengono estratte le prime informazioni essenziali per il resto del flusso, tra cui il sistema operativo e il tipo di file (PE, ELF, ...).

Già da qui, se il file non viene rilevato come PE o ELF, andiamo a saltare l'esecuzione di `capa` che sicuramente condurrà ad un errore. Per coprire anche tutti gli altri casi precedentemente discussi, dobbiamo eseguire l'analizzatore statico costruito ad-hoc (sez. 2.3). Questo andrà a controllare con molta più accuratezza se è un tipo di eseguibile supportato, controllando sia le situazioni descritte da alcune regole capa che l'architettura dato che è compatibile solo con x86 e x86_64.

Se ci troviamo in una situazione compatibile, viene eseguito `capa`. Questo è lo strumento più potente ma anche il più lento ed esoso di risorse, sia in termini di spazio che di tempo. Siccome è il più probabile al crash, viene gestito il suo exit-code fallimentare con un error message diverso dagli altri, per restituire al termine dell'esecuzione più informazioni possibili di debug e diagnostica, oltre ai log che però non sono direttamente inclusi nel file di risultato finale di analisi, ma dovranno essere collezionati separatamente. Come annunciato, l'output testuale di capa viene inviato al parser con una pipe, il quale darà in output il JSON risultante.

Dopodicché si prosegue con l'esecuzione dell'altro importante strumento: `yara`. Offre la possibilità di indicare a runtime quali set di regole eseguire. Ricordiamo di aver provveduto alla divisione delle regole collezionate in collezioni per sistema operativo, quindi leggendo il sistema operativo precedentemente identificato, possiamo specificare quale collezione usare. In caso non si tratti di un eseguibile o sia un sistema operativo sconosciuto / non considerato, verranno eseguite tutte le regole in quanto i rischi di perdere informazioni utili in questo caso supererebbero i benefici del risparmio di regole.

Infine, procediamo con l'esecuzione di strumenti quali ExifTool e fuzzy hashing con ssdeep. Tutti gli strumenti avranno così dato in output i propri risultati nel proprio file JSON. In conclusione, viene invocato un altro script Python che si occuperà di eseguire una validazione sintattica di tutti i risultati e, in caso di successo, eliminerà i file parziali per restituire l'unico report JSON complessivo di tutto. Questo ultimo layer di astrazione è utile sia per unificare i risultati intermedi

che per avere la possibilità di modificare gli strumenti sottostanti con analoghi, potendo sfruttare questo ultimo passaggio per uniformare il formato dell'output così da rendere il tutto trasparente al chiamante.

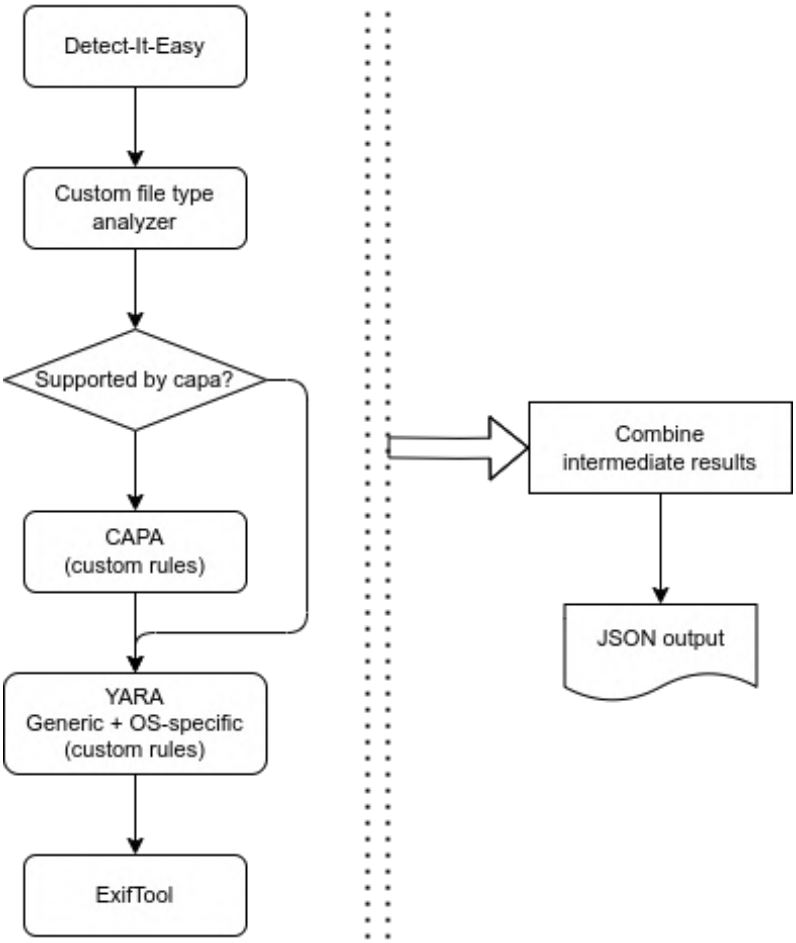


Figura 2.14. Workflow processo interno di `run_analysis`

2.7.2 Efficienza delle regole

Noto che ad ogni esecuzione di un'immagine Docker all'interno di un ambiente containerizzato, il container finale verrà distrutto, ci si è resi conto di come alcune regole vedano sempre una precompilazione o caching ad ogni invocazione, quando sarebbe idealmente più ottimale eseguire questo step una sola volta. Dopo uno studio più approfondito di questi strumenti, è emerso come i due principali (ossia `capa` e `yara`) abbiano modi per eseguire questo step in anticipo.

Per quanto concerne capa, successivamente aver analizzato il proprio processo di CI/CD [7] disponibile nel repository GitHub da cui è possibile scaricarlo precompilato od ottenere il sorgente, ci si è resi conto che questo processo sarebbe stato realizzabile.

[7] `https://github.com/mandiant/capa/blob/master/.github/workflows/build.yml`

Nel processo di build della nostra immagine Docker viene ora inserito un passaggio che permetta di:

- Clonare il repository GitHub di capa
- Eseguire il checkout del repository e dei propri submodules alla versione fissata, così da rendere la build al massimo replicabile in futuro
- Inserire le regole personalizzate, rimuovendo le predefinite
- Eseguire un caching delle regole ahead-of-time, come nel processo ufficiale di CI/CD
- Ora poter compilare il nostro eseguibile di capa e usarlo nelle varie esecuzioni al posto del precompilato già disponibile

In secondo luogo, occupandoci anche del tool `yara`, si è visto come disponga di API Python che ne permettono l'uso. Seguendo la documentazione ufficiale, offre la possibilità di salvare su file la versione precompilata delle regole. Così, analogamente a quanto appena visto, è stato inserito uno step aggiuntivo nella build che prenda le regole personalizzate, esegua queste funzioni da uno script Python, salvando i file precompilati senza dover conservare anche le regole originali, avendo tuttavia cura di mantenere la struttura delle cartelle inalterata.

Queste modifiche hanno permesso di rendere le esecuzioni più veloci di un tempo al momento trascurabile, ma che porterà maggiori benefici in futuro, quando il numero delle regole diventerà sostanzialmente più elevato, eseguendo uno step solo al momento di build e non ad ogni esecuzione.

2.7.3 Multi-stage build

Rimanendo in tema di build dell'immagine Docker, ci si rende subito conto come l'immagine prodotta in maniera naive abbia dimensioni di > 2 GB. Essendo una dimensione non giustificabile dalla presenza di strumenti troppo pesanti, si è reso necessario trasformare il Dockerfile per utilizzare una metodologia multi-stage. Questa tecnica permette di generare e lavorare su tante immagini diverse, ad esempio per eseguire compilazioni, copiare codici sorgente, installare software di compilazione pesante poi mai più usato in fase di esecuzione, per poi distribuire solo l'ultima di questa serie di immagini, che dovrebbe ottimalmente contenere solo lo stretto indispensabile per una corretta esecuzione, rimuovendo il superfluo che resta nelle immagini intermedie non distribuite.

Dividiamo il processo di build in vari step logici, avendo sempre cura di fissare l'hash delle immagini usate dal registry Docker Hub, così da avere la certezza di poter riprodurre la build anche in futuro e anche in caso di modifica/aggiornamento delle immagini corrispondenti a un dato tag:

1. Partendo da una immagine Debian 12, installiamo python, pip e tutte le altre dipendenze richieste a tempo di compilazione o utilità come git
2. Nell'immagine di Python così ottenuta, creiamo un nuovo virtualenv e impostiamo le necessarie variabili d'ambiente; riguardo al virtualenv ricordiamo

come non sia possibile eseguire lo script `.venv/bin/activate` come si farebbe normalmente su un sistema interattivo, perché ogni comando `RUN` in un Dockerfile esegue su una shell diversa, quindi dobbiamo modificare `PATH` e `PYTHONPATH` per puntare all'interno del virtualenv

3. In un'immagine, andiamo a clonare capa da GitHub, eseguire il checkout alla versione desiderata attraverso il tag di git e costruire l'eseguibile come precedentemente illustrato

4. Parallelamente, creiamo un'immagine che conterrà molte più dipendenze di build su cui andremo ad installare yara ed eseguire la precompilazione delle regole

5. Ripartendo da un'immagine Debian 12 slim pulita, installiamo solo Python e le librerie strettamente necessarie a runtime

6. Su questa, installiamo i tool secondari, come Detect-It-Easy o ExifTool che non richiedono passaggi aggiuntivi se non il download di un file

7. In conclusione, verrà copiato il codice sorgente del progetto realizzato e distribuita quest'ultima immagine

8. Esiste anche uno step di test preliminare, dove viene controllato che i comandi esistano nel `PATH` e poco altro, visto che i test funzionali veri e propri non sono inclusi nel Dockerfile, come dovrebbe essere da best practises

Possiamo notare in figura 2.15 come il processo di build ad un certo punto si divida tra la precompilazione di capa e di yara. Sfruttando il più recente comando `docker buildx` al posto dell'ormai deprecato `docker build` possiamo ridurre i tempi di build dell'immagine finale di vari minuti, variabili a seconda del sistema su cui si esegue la build, ma pur sempre un gran risparmio di tempo ottenuto grazie all'effettiva parallelizzazione dei due branch.

2.7.4 Entrypoint e flessibilità

Come già annunciato, una delle linee guida mantenute a mente per l'intero progetto è quella della massima flessibilità degli strumenti creati. Qui vediamo una sua prima applicazione pratica. Un'immagine Docker ha due proprietà distinte ma poco frequentemente adoperate: `ENTRYPOINT` e `CMD`. L'entrypoint è quel comando che viene sempre eseguito all'avvio di un'immagine Docker e permette di eseguire una prima inizializzazione, come creare qualche cartella mancante, fare dei check di avvio essenziali o cambiare utente. Di default, un'immagine ha entrypoint `/bin/sh -c` e nessun comando specificato. Il comando realmente eseguito è l'entrypoint, ricevendo come argomento il CMD. Com'è subito possibile notare dal flag `-c`, nel caso del default entrypoint, questo non farà altro che eseguire i comandi usando la shell `/bin/sh`.

Nel caso specifico di questo progetto, la divisione tra entrypoint e cmd viene notevolmente in aiuto nei casi in cui ci aspettiamo di avere vari utilizzi (implementati attraverso più cmd) ma vogliamo raccogliere a fattor comune le prime inizializzazioni obbligatorie (un unico entrypoint). L'entrypoint ora costruito, andrà ad eseguire

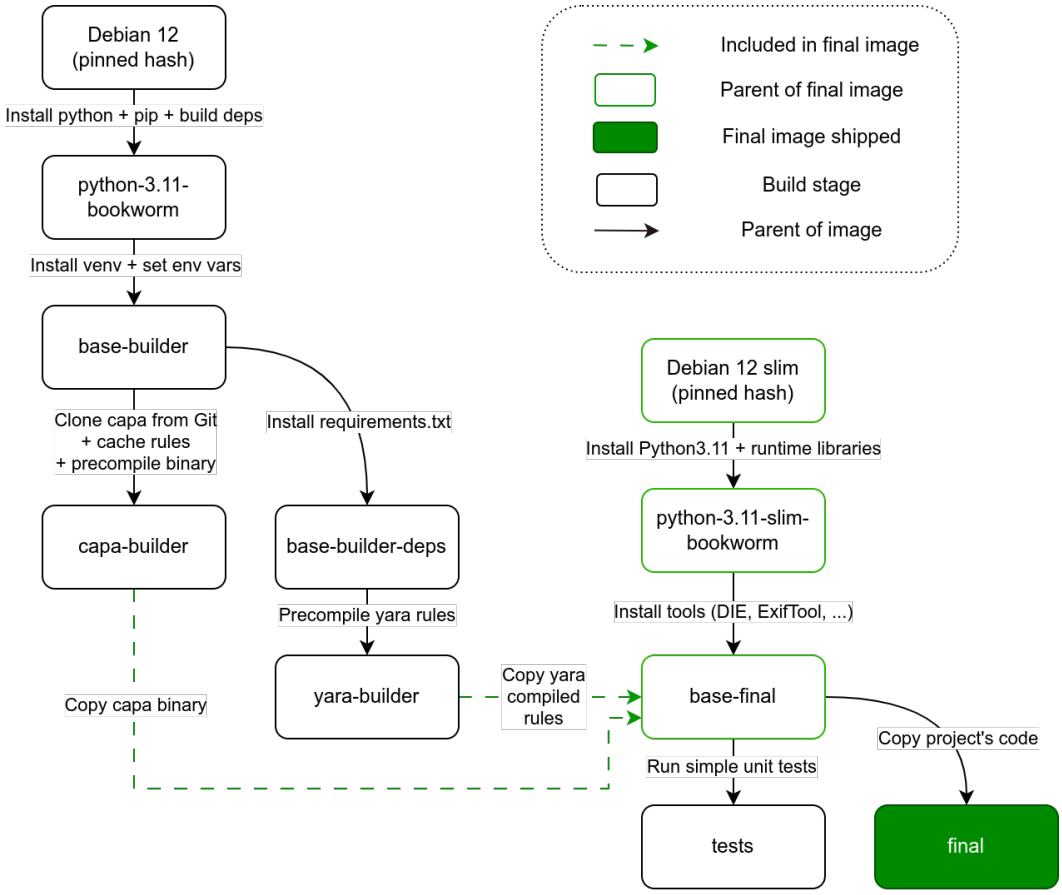


Figura 2.15. Docker multi-stage build, con le immagini intermedie

varie operazioni di bootstrap come controllare se sta eseguendo all'interno di una runtime AWS Lambda o su un sistema Docker tradizionale, creare la cartella di output se non esistente e proseguire l'esecuzione con un utente non privilegiato, siccome su un ambiente Docker tradizionale è comune che vengano eseguite come root, non gradito nel nostro ambito. Seppur non è come eseguire come root nel sistema host, essere root in un container è comunque un eccesso di privilegi rispetto al minimo indispensabile. Ciò che deve essere eseguito è definito dal CMD che andrà a ricevere come argomento (essendo uno script, lo leggerà nella variabile `"$@"`).

Al momento, sono realizzati vari script da usare come cmd a seconda del caso d'uso specifico:

- `analyze_multiple.sh` si aspetta di ricevere l'insieme dei file da analizzare nella cartella montata nella posizione di input, procedendo a scrivere i JSON risultanti nella cartella di output seguendo la struttura di directory dell'input;
- `download_and_analyze.sh` si aspetta di ricevere come argomento un URL da cui scaricare con `wget` il sample da analizzare - è utile sia nei casi in cui non vogliamo far transitare il sample sul filesystem del sistema host, ma anche nel caso di AWS come vedremo nella prossima sezione

Il cmd viene specificato agevolmente al momento dell'esecuzione e corrisponde agli argomenti passati al comando `docker run` successivamente al nome dell'immagine. A titolo esemplificativo, se il comando eseguito è: `docker run -it -rm -v /home/simone/output:/app/output static-engine:latest /app/src/download_and_analyze.sh https://example.com/malware.bin` lo script di entrypoint verrà eseguito ricevendo come argomenti il cmd, composto dallo script di download e analisi, nonché dall'URL del malware, che sarà successivamente l'argomento ricevuto da quest'ultimo script.

2.8 Automazione su AWS

2.8.1 Architettura serverless

Per architettura serverless si intende un particolare tipo di soluzione ingegneristica dove si procede a creare ed eseguire software senza l'onere di gestire l'infrastruttura sottostante, come ad esempio gestire il proprio server fisico, configurarlo e mantenerlo aggiornato nel tempo. Questo permette un risparmio di tempo ed è meno error-prone se non si hanno figure professionali specializzate dedicate. Ad oggi è generalmente la soluzione da preferire nella maggioranza delle situazioni in termini di costo-opportunità. L'architettura è ben riassunta in figura 2.16 e si basa sul paradigma *FaaS* (Function-as-a-Service), di seguito spiegata, dove l'entità principale della computazione è una funzione.

Il flusso di esecuzione ha inizio con l'upload del sample su un bucket S3 [8] che ha lo scopo di archiviare tutti i sample da analizzati o che sono stati analizzati, per costruire un proprio repository interno di sample, maligni o meno che siano. Su questo

[8] S3 è un servizio di AWS che si occupa di object storage, un tipo di storage di file dove ciascuno è visto come un nome con il proprio contenuto in byte, metadati e poche altre informazioni - il nome S3 è acronimo di Simple Storage Service

bucket è previsto che proceda al caricamento il componente aziendale FSL, parte dell'area di Threat Intelligence. Tuttavia, siccome non è ancora sviluppato ma è solo in programma, a fini di test e utilizzo iniziale, l'upload viene effettuato manualmente dalla console AWS, visto che non comporta alcuna differenza nel resto del processo. L'upload andrà così ad eseguire il trigger dell'evento `ObjectCreated:Put`, ascoltato dalla prima delle 3 lambda coinvolte. Bisogna prestare attenzione al fatto di dover ascoltare sia l'evento Put che `ObjectCreated:CompleteMultipartUpload` visto che in caso di caricamento di file grandi, verrà invocato solo l'ultimo evento. Se non fosse ascoltato, non verrebbe mai scatenata la prima lambda e, di conseguenza, tutto il resto dell'analisi verrebbe meno.

La scelta architetturale di dividere l'esecuzione in 3 lambda anziché lasciar svolgere tutti i task a una sola Lambda, risiede nel voler minimizzare i permessi concessi alla fase di analisi. Seppur si tratta di un processo statico, quindi non prevede esecuzione del sample, non è in alcuna maniera escludibile una vulnerabilità di tipo RCE (Remote Code Execution) all'interno degli strumenti adoperati, o che verranno aggiunti successivamente, ma è anzi più frequente di quanto ci si possa aspettare, e la superficie di attacco tende ad aumentare in maniera direttamente proporzionale al numero e alla complessità dei tool adoperati.

Al contrario, con questa soluzione, è stato possibile ridurre i permessi della fase di analisi, dove il contenuto del sample viene interpretato e non semplicemente trasferito sotto forma di byte o URL, ritenuto il momento più critico nell'intero processo. Può solamente:

- Scrivere nelle proprie tracce di log su CloudWatch
- Invocare unicamente la Lambda a sè successiva
- Leggere solo il sample che deve analizzare

Non può quindi:

- Accedere al bucket S3 per intero, prevenendo fughe di dati
- Accedere al servizio della Threat Intelligence che conserva i dati sulle analisi e i loro risultati

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "logs:PutLogEvents",
            "logs:CreateLogGroup",
            "logs:CreateLogStream"
        ],
        "Resource": "arn:aws:logs:*:*:*"
    },
    {
        "Effect": "Allow",
```

```
        "Action": [ "lambda:InvokeFunction" ],
        "Resource": "<static_analysis_reporting ARN>"
    }]
}
```

Listing 8. JSON della policy assegnata alla Lambda dell'analisi

Le policy assegnate alle varie entità su AWS sono state scritte da zero, mantenendo chiara la politica del minimo permesso concesso; sono state ritenute eccessivamente permissive le policy gestite fornite direttamente da AWS.

La prima Lambda, chiamata `trigger_new_malware`, si occupa di ricevere l'evento invocato al termine dell'upload, quindi estrarre quali siano i nuovi file caricati, per ciascuno generare un **pre-signed URL** così da permettere la lettura del singolo oggetto per un periodo di tempo limitato e senza concedere permessi di lettura del bucket a terze entità. Andrà quindi a invocare in maniera asincrona la lambda di analisi, inviando nel payload l'URL appena generato e l'ID dell'analisi. L'ID è uno UUIDv4 generato in maniera pseudo-random e serve per tenere traccia dell'analisi durante la propria esecuzione, sia nei log che nel reporting finale per stabilire con certezza quale sia l'analisi a cui assegnare i risultati. Non appena l'analisi ha inizio, verrà informato l'ipotetico servizio di Threat Intelligence riguardo l'avvio con successo di questo nuovo task.

All'interno della lambda di analisi viene aggiunto un nuovo script da usare come cmd (`lambda_entrypoint.sh`): ha lo scopo di fornire l'interfaccia runtime Lambda necessaria per l'esecuzione in tale runtime. Non è possibile eseguire direttamente una immagine Docker così com'è su AWS Lambda, ma necessita di esporre una precisa interfaccia per compatibilità. Per rispettare questo requisito si può scegliere tra usare i container forniti da Amazon come base o usare altri strumenti. Nel nostro caso, trattandosi di un container molto poco straightforward, si è optati per la seconda opzione. È stato sufficiente invocare il modulo `awslambdaric` per eseguire uno script Python che espone una funzione avente la stessa interfaccia che ci si aspetta da una Lambda standard Python, ossia una funzione handler che accetti due parametri: l'evento e il context. Ci interessa principalmente l'evento, contenente il payload ricevuto durante l'invocazione.

Infine, per permettere anche un test della Lambda all'interno di una più simile runtime possibile all'ambiente di produzione, si sfrutta `aws-lambda-rie`, un tester con cui è possibile simulare l'ambiente finale eseguendo in locale.

2.8.2 Limiti delle Lambda

È bene sottolineare come le Lambda non siano lo strumento di computazione adatto in tutti i casi. In questa sezione vengono esposti i contro e le limitazioni che comportano, nonché i rimedi adottati per limitarne gli effetti.

Una delle prime limitazioni riguarda le risorse concesse dal server in termini di potenza di calcolo. Viene imposto a tutte le invocazioni un timeout estendibile fino a 15 minuti, e la potenza della CPU assegnata è proporzionale alla memoria richiesta. È fondamentale essere consapevoli di tali caratteristiche del servizio, così da prendere contromisure quanto più efficiaci possibili.

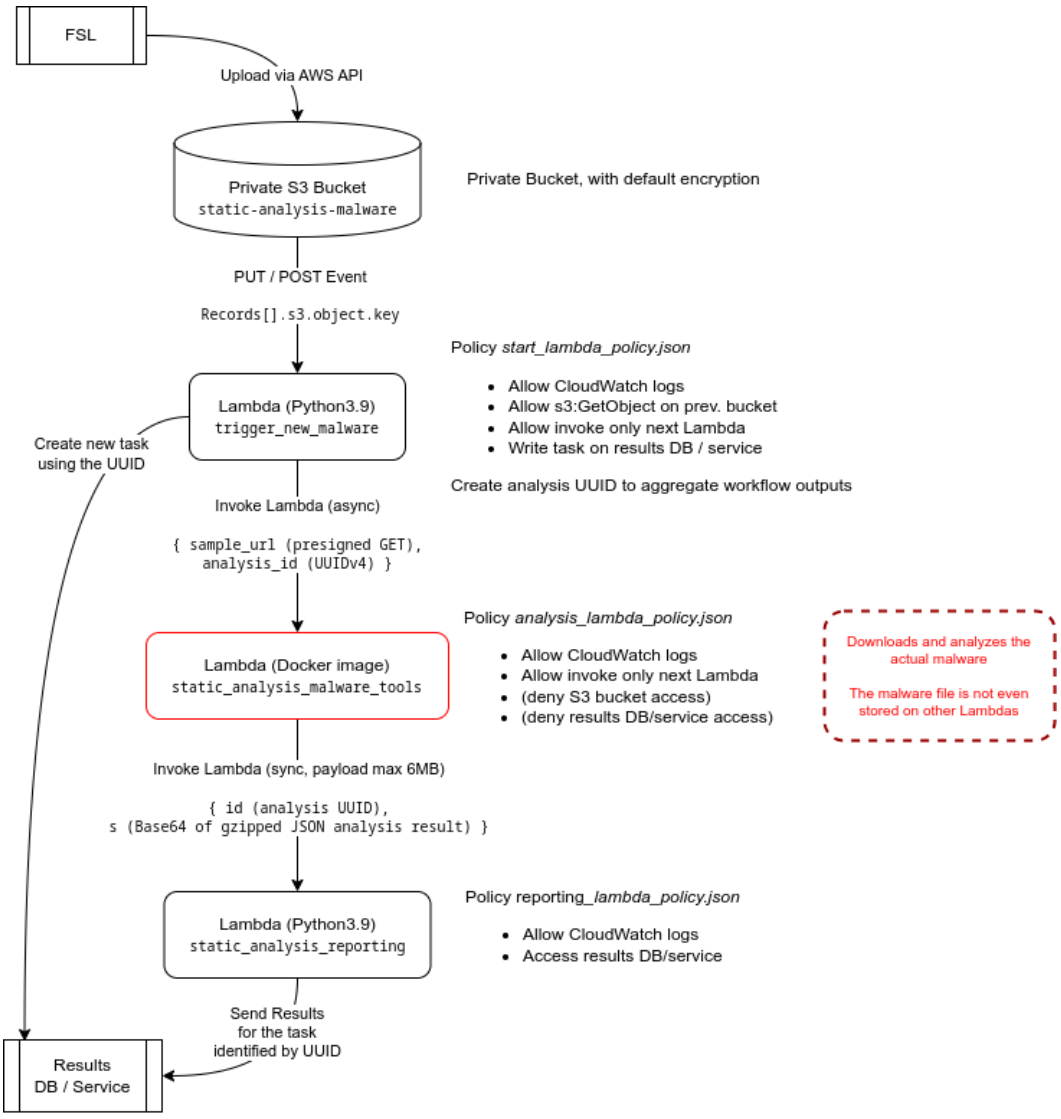


Figura 2.16. Architettura su AWS con i vari componenti coinvolti

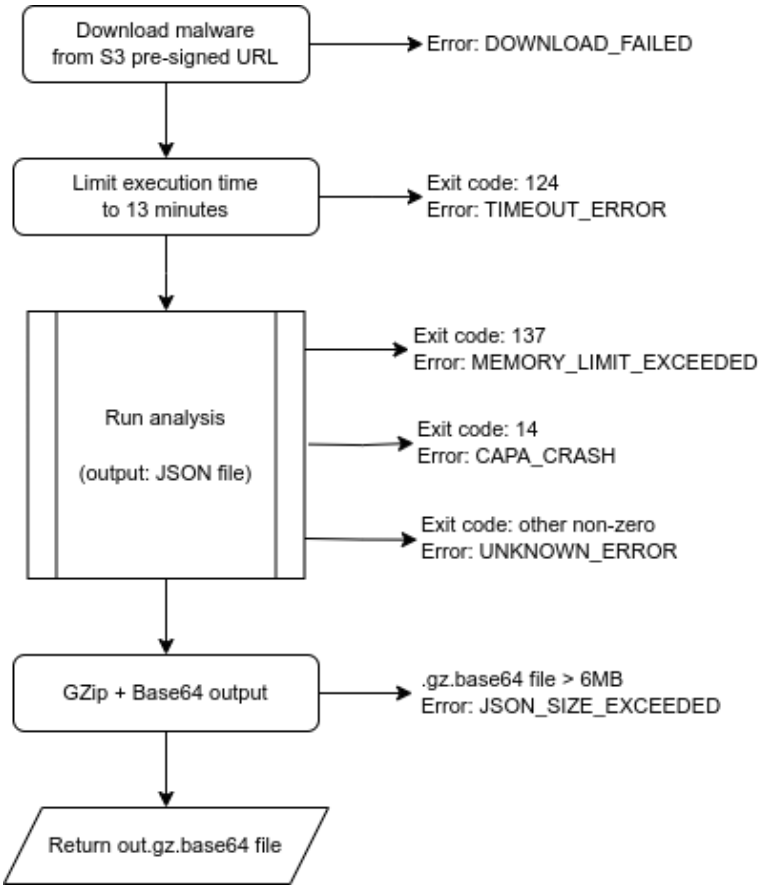


Figura 2.17. Workflow dell'analisi in `static_analysis_malware_tools` e possibili errori

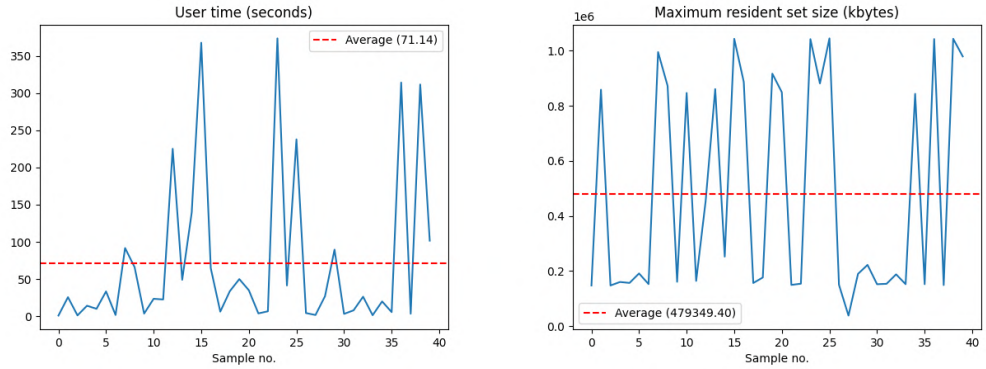
Prima di tutto, si procede a misurare il tempo di esecuzione e la memoria richiesta da ogni analisi così da avere dei dati ottenuti da misurazioni su cui lavorare. Nell'immagine Docker precedentemente realizzata, si va ad eseguire il processo di analisi attraverso `/usr/bin/time`, da non confondere con il comando `time` generalmente integrato nella shell - sono distinguibili ed elencabili eseguendo il comando `type -a time`. L'eseguibile GNU, con il flag `-v`, permette di misurare sia il tempo di esecuzione che la memoria totale richiesta, consentendoci di eseguire delle stime. L'output verrà così salvato su un file testuale per poi essere analizzato, traendone più informazioni dai dati ottenuti.

Successivamente, eseguiamo il container così modificato su un sample set composto da 40 eseguibili scelti casualmente da VirusTotal, estraendo output e misurazioni. Infine, raccogliamo i dati in un grafico (figg. 2.18a e 2.18b) per visualizzare il tempo e la memoria, tracciando una linea per rappresentare il valore medio. Notiamo come il tempo medio di esecuzione sia di circa 71 secondi, di gran lunga inferiore al limite di 15 minuti, ossia 900 secondi, per invocazione. Tuttavia, seppur non particolarmente grave, abbiamo appena citato come la potenza computazionale attesa sull'ambiente Lambda sia proporzionale alla memoria massima stabilita come allocabile e comunque inferiore a un ambiente di esecuzione locale riservato solo a questo scopo, per cui è da usare tali stime solo come un punto di riferimento vago, e non sottovalutare casi in cui questo possa aumentare fino a superare il timeout. Proprio per ciò, viene imposto un limite già all'interno del container di esecuzione a 13 minuti per l'analisi, usando il comando `timeout` della shell, così da avere il tempo per permettere una situazione di *graceful exit* e portare sempre a un output, che in questo caso conterrà il flag di successo a `false`, assieme al campo `"error"` valorizzato.

Per quanto concerne l'utilizzo di memoria, ci si attesta attorno ai 500MB circa di consumo, seppur non segua affatto una distribuzione normale, visto come ci siano sample che richiedano molta più memoria e altri molta meno, quasi nessuno quanto la media. Anche per questa casistica bisogna tenere conto dei casi di fallimento e prevedere una *graceful exit*. Siccome si è visto che la quasi totalità dell'uso è dovuto dall'esecuzione di `capa`, si è voluto osservare il suo comportamento in caso di ragggiungimento del limite di memoria disponibile. Perciò, sfruttando il flag `-memory` di Docker run, si è impostato un limite molto basso e si osservano i risultati. Si è subito rilevato come ciò comportava un crash del tool specifico con exit code 137, associato proprio alla condizione di OOM (Out-Of-Memory). Il flusso di esecuzione però poteva continuare l'esecuzione, a patto di gestire l'errore, dovuto dal fatto che all'inizio dello script di analisi è stato impostato il flag `set -e` che comporta un'uscita automatica appena un comando dovesse restituire un exit code non-zero. La soluzione adottata è quella di gestire l'errore nello script chiamante, a livello più alto, quindi interrompendo l'analisi ma dando in output l'errore in maniera analoga a quanto indicato per il precedente caso del timeout.

Questa gestione degli errori porta il vantaggio di avere in ogni caso in output un JSON valido, sempre presente e in grado di fornire un'indicazione dell'errore al processo invocante (nello scenario finale sarà presumibilmente la Threat Intelligence).

Non si è ritenuto necessario procedere con l'upload dei risultati in un bucket S3, nè é ragionevole concedere alla Lambda di analisi il ruolo di contattare il sistema di reporting, che sia l'API della Threat Intelligence o un mock temporaneo, per i



(a) CPU **(b)** Memoria

Figura 2.18. Uso delle risorse eseguendo analisi statica su malware reali presi a campione

motivi di sicurezza precedentemente elencati. L'unica soluzione restante è quella di inviare il risultato come payload alla Lambda responsabile di quest'ultimo step. Tuttavia, AWS impone certi limiti per questa operazione: in caso di invocazione asincrona non può superare i 256KB, aumentabile fino a 6MB andando a usare l'invocazione sincrona. Quest'ultima ha lo svantaggio di addebitare il proprio tempo di esecuzione anche alla Lambda chiamante, ma non essendo un'operazione più lunga di pochissimi secondi, è un compromesso che si è disposti ad accettare.

Presa conoscenza di tale limite aggiuntivo, si procede ad effettuare una misurazione della dimensione in byte del JSON di output, già minimizzato e non formattato. Siccome è subito intuibile come questo possa essere troppo spesso oltre questa dimensione, si implementa una strategia per ridurne la dimensione. Una prima idea poteva essere l'uso di una struttura *"dizionari di liste"* al posto della più tradizionale *"liste di dizionari"* comunemente adottata. Tuttavia, è stata scartata perché, seppur avrebbe fatto risparmiare la ripetizione delle stringhe corrispondenti alle chiavi dei dizionari interni, avrebbe comportato un'eccessiva trasformazione e molta meno libertà di futura modifica del formato o l'aggiunta/rimozioni di alcune sue parti.

La soluzione realmente adottata è stata usare `gzip`, un comune strumento di compressione che usa l'algoritmo *DEFLATE*, usato con il massimo livello di aggressività, per ridurre in maniera sostanziale lo spazio occupato dal JSON, sfruttando le sue proprietà come l'essere un file testuale, avere spesso ripetizioni come nel caso delle chiavi dei dizionari e altre caratteristiche che ne favoriscono la comprimibilità. Infine è necessario svolgere un ulteriore passaggio, visto che il payload delle Lambda non può essere uno stream di byte ma è richiesto che sia un oggetto JSON valido, quindi contenente testo. Viene preso il payload in byte ottenuto come risultato della compressione, e trasformato in testo adottando la codifica *Base64* che però incrementa la dimensione di circa il 33%, incluso in un JSON minimale avente solo la chiave `"s"` con all'interno questa codifica.

Dopo aver implementato questa strategia, è stata eseguita una misurazione paragonando la dimensione originale del JSON originale, del JSON codificato come appena illustrato e il livello dei 6MB come limite di ciò che è possibile inviare (fig.

2.19). Si nota immediatamente come la dimensione della linea della versione codificata sia notevolmente inferiore a quella del file originale, abbattendo le probabilità di andare oltre il limite. Siccome improbabile non significa impossibile, è stato d'obbligo prevedere una graceful exit anche in tale evenienza.

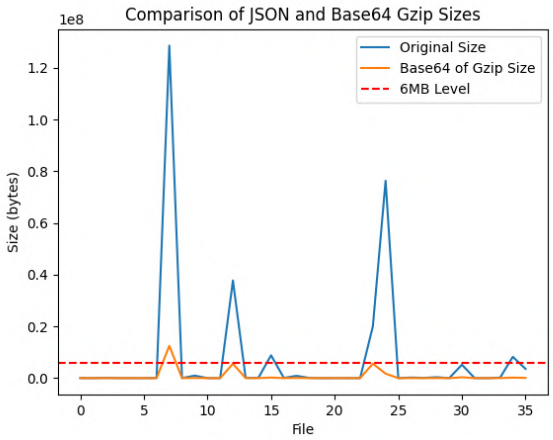


Figura 2.19. Dimensione del file di output originale e post-compressione gzip + base64

Per riassumere tutti i casi di errore evidenziati, è stata definita e documentata una lista chiara dei codici di errore che possono verificarsi, ossia:

- `TIMEOUT` per i casi di timeout, riconoscibili dall'exit code del comando timeout uguale a 124
- `MEMORY_LIMIT_EXCEEDED` attribuito all'exit code 137 (OOM) in caso di superamento del limite di memoria
- `DOWNLOAD_FAILED` se nello script che accetta un URL come parametro e non direttamente, la richiesta HTTP è fallita sia per problemi di connessione che di status code della risposta non 2xx
- `CAPA_CRASH_UNKNOWN` assegnato al crash di capa, strumento più delicato di tutto il processo, diverso da motivazioni già precedentemente gestite
- `JSON_SIZE_EXCEEDED` nei casi di un output JSON riuscito con successo ma eccessivamente grande da superare i limiti di payload di invocazione delle Lambda
- `UNKNOWN_ERROR` paragonabile a una risposta HTTP 500 Internal Server Error di un webserver, dove non è ben chiara la causa del crash ed è bene controllare i log
- `FATAL_SCRIPT_CRASH` nei casi più estremi in cui lo script di analisi non riesce a generare nè un successo nè un errore nel JSON di output, allora l'ultimo attore rimasto presente è l'handler Python della Lambda, che procederà ad emettere questo errore; si presuppone non succeda mai ma è conservato come ultimo tentativo di emettere un errore, in caso si verifichino situazioni molto più particolari non previste

La verifica che questi siano gli unici e i soli codici di errore restituibili in output è data dal sistema usato per creare il JSON di errore: uno script Python che da argomento da terminale prende in input l'errore che si vorrebbe restituire, controlla che sia tra quelli consentiti o ne restituisce uno di default segnalando l'avvenimento fuori dal protocollo definito nei log, per poi creare il JSON con il formato standard solo in caso non sia già stato dato un altro errore per la stessa invocazione. Quest'ultimo può essere rappresentato dal caso in cui la propagazione dell'exit code fallisca e la seconda volta venga richiesta l'emissione di un errore meno specifico, che lo script Python andrà a rifiutare.

2.8.3 Costi del servizio

Le prossime analisi si baseranno sui seguenti parametri di riferimento forniti sulla base dello storico aziendale e delle aspettative di uso reale del servizio:

Parametro	**Valore**
Numero di sample da analizzare	100.000 al mese
Memoria media richiesta	512 MB (limite massimo: 1024MB)
Tempi medio di esecuzione	70 secondi

Inoltre, si sottolinea come i costi dei servizi AWS subiscano inflessioni nel tempo, per cui questa stima è sufficientemente valida al momento della scrittura di questa relazione. Per i calcoli viene usato il calcolatore ufficiale Amazon[9] inserendo come posizione Milano (eu-south-1) per ragioni di GDPR, policy aziendali e distanza dal server che effettuerà l'upload dei sample. In merito al numero di GET stimate al mese, si considerano, per ogni sample, 1 GET per la lettura nell'analisi e il 20% delle volte una GET su Internet per il download del file dall'analista. Viene escluso dai calcoli, l'uso del Free Tier, in quanto questo potrebbe essere utilizzato per ulteriori progetti sullo stesso account.

Dimensione media di un sample	1 MB
Richieste PUT al mese	100.000
Richieste GET al mese	120.000
Numero di sample dopo 3 mesi	300.000
Spazio occupato dai sample dopo 3 mesi	300.000 MB $\approx$ 280 GB
Costo stimato mensile	€ 10,00

Tabella 2.1. Parametri per il costo di un bucket S3

[9] `https://calculator.aws`

Lambda trigger post-upload	
Durata di esecuzione media	2 secondi
Memoria allocata	256 MB
Costo stimato mensile	€ 1,00
Lambda analisi con strumenti	
Durata di esecuzione media	70 secondi, limitato a 14 minuti
Memoria allocata	1024 MB
Costo stimato mensile	€ 124,00
Lambda reporting	
Durata di esecuzione media	5 secondi
Memoria allocata	512 MB
Costo stimato mensile	€ 5,00
Invocazioni di tutte le Lambda	100.000 al mese per Lambda

Tabella 2.2. Parametri per il costo delle varie Lambda

Servizio	**Stima**
S3 Bucket (standard tier)	€ 10,00
Lambda start	€ 1,00
Lambda di analisi	€ 124,00
Lambda di fine rapporto	€ 5,00
Costo stimato totale al mese	€ 140,00

Tabella 2.3. Riepilogo dei costi stimati per i singoli servizi

Viste le spese per task analoghi già sostenute dall'azienda, è ritenuto un valido preventivo, anche in ottica di futuro reale utilizzo del progetto all'interno del più vasto sistema.

Sono state tuttavia valutate anche altre opzioni, come l'uso di una o più istanze EC2 pre-allocate, con posto di fronte un load balancer, nel tentativo di ridurre i costi. Tuttavia, non è stato efficace ma controproducente, andando anche a perdere le qualità delle Lambda quali l'auto-scaling e il costo nullo in caso di non utilizzo.

2.9 Versioning del software

Al fine di mantenere in maniera ordinata il software realizzato e la relativa documentazione, è stato creato un nuovo repository Git all'interno del server GitLab aziendale. Questo è stato unicamente dedicato al progetto. Come da utilizzo standard di Git,

per ogni modifica viene realizzato un commit. Si adottano tuttavia una serie di convenzioni atte a tenere alta la qualità e la mantenibilità del codice in questione:

- I commit devono essere atomici, ossia contenere una ed una sola modifica logica realizzata, per migliorare la comprensione quando in futuro vengono riletti e permettere un eventuale revert efficace, se non fare cherry-picking di un commit da un branch all'altro
- I messaggi dei commit devono essere quanto più descrittivi possibili, evitando in maniera assoluta descrizioni vuote o non significative, come *"Update"* o *"Fix stuff"*
- Idealmente, i messaggi dovrebbero iniziare con un verbo all'infinito, indicando l'azione svolta
- L'autore deve essere ben chiaro, inserendo un Co-Author se necessario e potenzialmente adottando sistemi di firma del commit con una chiave personale GPG, per essere certi di attribuire la giusta paternità
- Il branch `main` (o `master`) è protetto: non è possibile eseguire force push, limitando operazioni potenzialmente distruttive sull'intera codebase
- I branch seguono nomi standard, illustrati nella sez. 2.9.1

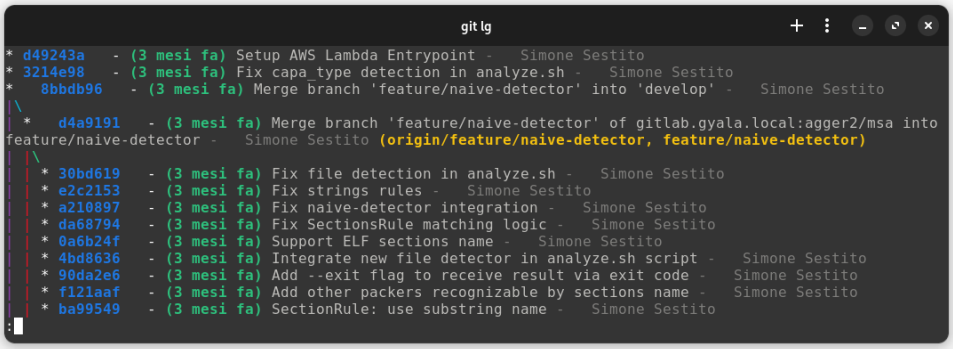


Figura 2.20. Esempio di history Git con più branch

2.9.1 Branch Naming Convention

Per la gestione del repository,e in particolare i suoi branch, si seguono rigidamente queste convenzioni:

- `main` è il branch principale dove deve essere presente solo codice finito e considerabile stabile a tutti gli effetti;
- `develop` è dove si tiene il codice che si considera usabile ma non ancora sufficientemente testato o pronto all'uso in produzioni così com'è - su questo branch infine è dove viene fatto il merge dei successivi branch più specifici;

- `feature/<feat-name>` è l'insieme di tutti i branch, originati da *develop*, dove si lavora all'implementazione di una specifica funzione - ciò permette di mantenere il branch *develop* funzionante, privo di funzioni non completamente realizzate, e diminuisce le interferenze tra chi sta lavorando sulla stessa codebase ma su funzioni distinte tra loro - e il prefisso **feature/** identifica chiaramente che si tratta di un branch dove si lavora solo ed esclusivamente su una sola funzionalità;

- `refactor/<ref-name>` invece rappresenta l'insieme dei branch dove si fa, con diversi gradi di complessità, refactoring del codice - ad esempio, nel progetto è stato realizzato un branch di refactoring per passare dall'uso di `os.path` al modulo Python `pathlib` per una maggiore usabilità

Infine, per un maggiore controllo, il merge dai branch secondari (`feature/` e `refactor`) vengono fatti attraverso l'uso di *Merge Request* (o *Pull Request*, a seconda che si usa la nomenclatura di GitLab o di GitHub). In questo modo, è possibile interagire sul merge commentando o richiedendo revisioni di co-workers.

Non siamo ancora giunti al punto di rilasciare il progetto e attribuirgli una versione, ma in tal caso è ideale usare convenzioni standard come *Semantic Versioning* [10] , e usare tale stringa come **tag** sul repository Git.

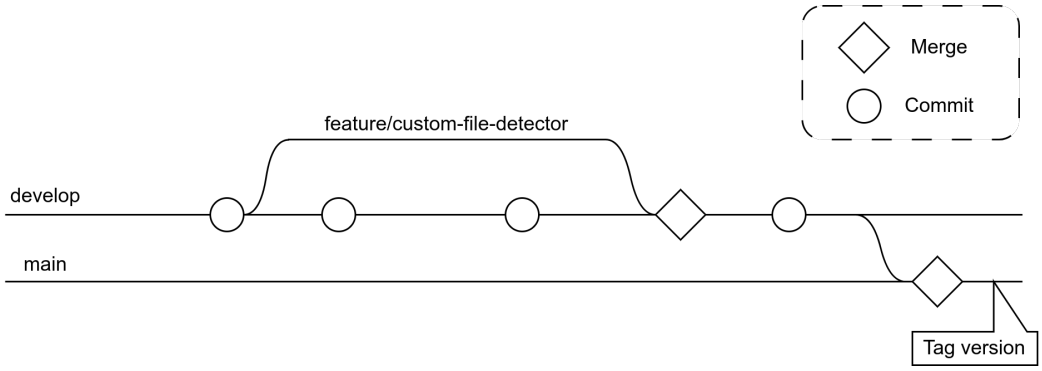


Figura 2.21. Struttura dei branch Git

2.10 CI/CD Pipeline

Successivamente alla fase di sviluppo, seguono il test e il deploy in ambiente di produzione o pre-produzione. Si è reso necessario automatizzare questo aspetto, per ridurre l'errore umano e i costi del personale per la gestione. La soluzione più idonea a questo scopo è stata sfruttare le funzionalità di pipeline del server GitLab aziendale. Infatti, è possibile definire tramite un file in formato YAML tutti gli step di build, test e deploy per il codice presente nel repository Git, eseguendoli su un Gitlab Runner.

Un GitLab Runner si concretizza in un server fisico che esegue questo strumento in grado di ricevere dall'orchestrator GitLab il prossimo task della coda che deve

[10] `https://semver.org/lang/it/`

eseguire, e riportare il risultato ottenuto, che sia un artefatto da tenere archiviato o un altro output. Esistono sia Runner forniti come SaaS (*Software-as-a-Service*), che self-hosted, come nel nostro caso. La potenza dell'orchestrator sta nell'astrarre tutto ciò, rendendo il processo egualmente funzionante a prescindere dalla reale infrastruttura sottostante. L'associazione tra il task della pipeline e i Runner disponibili ad eseguirlo avviene attraverso un tag, che sarà assegnato al primo nel file `.gitlab-ci.yml` e poi ai secondi nella pagina di configurazione della macchina. In questo modo, definiamo chiaramente quali server hanno la responsabilità per le pipeline di quali progetti, permettendo di definire anche ambienti distinti per le varie fasi, se il caso specifico lo richiede.

La pipeline realizzata per le operazioni di CI/CD nel progetto qui realizzato si compone dei seguenti step:

1. `build_docker`: fase di build dell'immagine Docker, partendo dal codice sorgente ottenuto dal repository e dalle istruzioni contenute nel Dockerfile, ottenendo come output l'immagine finale, pronta per essere testata. Verrà conservata per 2 settimane nell'archivio del server GitLab in caso servisse in un breve futuro fare ulteriori controlli;

2. `test_analysis`: fare di test, dove viene presa in input l'immagine appena prodotta e vengono eseguiti test funzionali. Questi hanno lo scopo di verificare che l'eventuale deploy di questa immagine non causi danni o regressioni al servizio, andando a controllare tutta una serie di casi tipici nei quali ci si può ritrovare, come un file eseguibile tradizionale, vuoto, di altro tipo o eseguibile ma non supportato da alcuni strumenti - vedi `capa` (sez. 2.3). Si limita anche il tempo di esecuzione, per prevenire casi di loop o tempo di esecuzione eccessivamente più lungo della versione precedente;

3. `aws_deploy`: fase finale dove avviene il deploy, se le precedenti hanno dato esito positivo. Il deploy dell'immagine Docker realizzata e testata come nuovo codice della AWS Lambda di analisi è implementato usando la AWS CLIv2, che permette di eseguire tale operazione nella maniera più semplice, prima caricando la nuova immagine sul container registry AWS ECR, poi andando a sostituire l'immagine della Lambda, senza creare downtime o disagi alle attuali esecuzioni in corso.

È bene notare come, in molti casi di self-hosting, anche il Runner stesso esegua all'interno di una sua immagine Docker. Dovendo però costruire, poi eseguire, un'immagine Docker all'interno di un altro container, è stato necessario adoperare il servizio *DinD* (Docker in Docker).

Inoltre, nella fase di test, i sample vengono presi direttamente da una cartella specifica nel repository Git, così da mantenerne la catalogazione in maniera semplice ed efficace. Tuttavia, in alcune configurazioni dei Runner, può accadere che DinD vada a richiamare il Docker daemon (`dockerd`) dell'host vero e proprio, e non uno virtualizzato nel container, come accaduto nel Runner aziendale già in opera per altri progetti. La soluzione qui è stata quella di **evitare i bind di cartelle con percorso assoluto** perché ciò avrebbe comportato una mal interpretazione del path, ponendolo come path dell'host mentre in realtà è del container stesso. In

alternativa è stato usato un nuovo volume Docker con un nome univoco per ogni esecuzione, così da spostare tutti i sample in esso attraverso un archivio Tar inviato tramite pipe, ed eseguito il bind di questo volume al container di test innestato anziché usare path assoluti, che com'è appena emerso, possono causare problemi in talune circostanze. Ottenere l'output dell'esecuzione sui sample di test segue la stessa logica ma in maniera speculare, pertanto è stata omessa per brevità.

Capitolo 3

Analisi dinamica

Per analisi dinamica di un malware si intende lo studio di tutti i suoi comportamenti durante l'esecuzione in un ambiente controllato, come una sandbox. Comportamenti sospetti possono essere la modifica di particolari chiavi di registri, la sovrascrittura di file privati dell'utente che non dovrebbe manipolare, la ricerca di file con nomi che potrebbero nascondere contenuti sensibili (ad esempio legge tutti i file che contengono la parola "password" nel nome, come `password.txt`) o altre situazioni ancora.

Lo scopo di ciò che verrà realizzato in questa fase del progetto è di analizzare automaticamente l'esecuzione di un malware, così da riportare tutti gli elementi degni di nota all'analista o al servizio di Threat Intelligence in un formato machine-readable come JSON. Le operazioni svolte in questa fase sono indipendenti dalla componente statica appena vista: infatti, l'obiettivo sarà di lasciare la decisione all'analista o alla Threat Intelligence automatica sulla base di una confidence del risultato finale (se malevolo o meno), per eseguire questo processo solo nei casi di reale utilità, siccome sarà molto più esoso di risorse, quindi avente un maggior costo economico.

L'analizzatore dinamico, comprensivo della sandbox e di tutto ciò che è descritto in questo capitolo, è stato sviluppato ed eseguito in locale. Tuttavia, è già stato predisposto per un futuro impiego su un server gestito dall'azienda dove possa eseguire e ricevere task attraverso una REST API o altro interfacciamento. Come nell'analizzatore statico, ritroviamo anche qui tutte le caratteristiche di qualità come la massima flessibilità ed estensibilità con altri strumenti nel corso della sua evoluzione.

L'esecuzione del possibile malware deve avvenire necessariamente in un ambiente isolato e supervisionato, idealmente riducendo a zero le possibilità che possa evadere e infettare il sistema host. Non potrà mai esistere un sistema che, in maniera assoluta, permetta l'esecuzione di un programma senza rischi, potendo esistere vulnerabilità zero-day in tutte le componenti coinvolte, senza imporre limitazioni che inficerebbero sull'esito dell'analisi stessa, come disconnettere fisicamente un host reale da tutte le interfacce di rete e distruggere l'ambiente completamente al termine. Un software predisposto per fare ciò prende il nome di *Sandbox*.

3.1 Cuckoo Sandbox

Per la realizzazione del progetto è stato adottato *Cuckoo Sandbox*. [1] Si tratta di un progetto open-source, rilasciato sotto licenza GPLv3 dalla Cuckoo Foundation, leader del settore che fornisce la base su cui lavorare e fare tutte le dovute modifiche del caso. Consente di venire a conoscenza dell'esatto comportamento avuto dal malware durante la sua esecuzione, includendo la lista completa delle chiamate di sistema di ogni processo, nonché la cattura del traffico di rete, screenshot della UI durante l'esecuzione e il contenuto dei file creati nel sistema.

Viene fornito di due componenti: l'*Host* e uno o più *Guest*. L'Host, anche chiamato Orchestrator, ha il compito di gestire i Guest, come avviarli, interromperli, analizzare il traffico di rete intercettato e tutta l'interpretazione dei risultati raw ottenuti dal Guest, così da ottenere il report. Un Guest è tipicamente una VM, ma può essere anche un host fisico distinto, che contiene il sistema operativo, tutti i software che ci aspetteremmo in un PC tradizionale che vogliamo simulare, insieme all'Agent di Cuckoo, che comunicherà con l'Orchestrator per coordinare il lavoro e riportare i risultati.

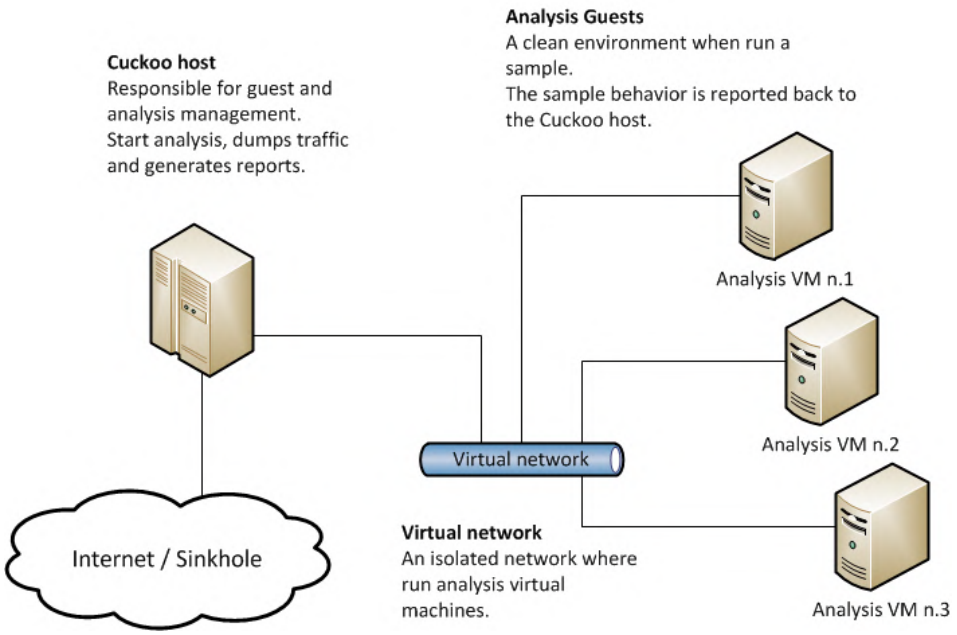


Figura 3.1. Architettura di Host e Guest di Cuckoo così come viene fornito
Crediti: cuckoo.readthedocs.io

Su questa base, sono state fatte numerose lavorazioni illustrate nelle sezioni successive di questo documento. Non meno importante, è stata eseguita tutta l'installazione, configurazione su misura e hardening.

[1] `https://github.com/cuckoosandbox/cuckoo`

3.2 Architettura realizzata

L'architettura finale che è stata realizzata, e che verrà dettagliata di seguito, si compone come in figura 3.2. Sono presenti diverse macchine virtuali:

- *Cuckoo Ubuntu VM* è l'Host, ossia dove risiede la sandbox e tutti i Guest. Espone, tramite un'interfaccia di rete locale di VirtualBox denominata *cuckoo-net*, una REST API dov'è possibile inviare sample da analizzare all'orchestrator e successivamente ricevere informazioni sullo stato del task, incluso il suo report, quando pronto per essere scaricato

- *API Client VM* è una macchina virtuale isolata dal resto del sistema che ha il compito di interagire con l'Orchestrator, inviando task e ricevendo le risposte. Fornisce un'interfaccia utente da linea di comando per l'interazione e leggerà, per comodità, i file da analizzare da un volume montato in sola lettura

- *Windows 7 Nested VM* rappresenta uno dei due Guest identici creati all'interno della Cuckoo VM, e si tratta di macchine virtuali VirtualBox innestate, con installato e configurato Windows 7 e tutti i programmi tradizionali, come ci si aspetterebbe di trovare un reale PC; di questo e altri accorgimenti per rendere la simulazione più fedele possibile ne verrà discusso nella sezione 3.2.3

L'uso di una VM che al suo interno ne contiene altre 2 o più innestate, anziché installare tutto direttamente sul server o computer vero e proprio, vede le sue ragioni nel voler aumentare il più possibile l'isolamento tra il sistema di sandbox e il computer dove ciò viene eseguito, nonché rendere il sistema nella sua interezza più portabile. Sarà necessario esportare le 2 macchine virtuali più esterne e creare le interfacce di rete per poter eseguire questo sistema altrove con la minima configurazione necessaria. L'unico requisito che viene aggiunto è il supporto alla virtualizzazione innestate (e non solo la virtualizzazione tradizionale) sul server fisico che eseguirà il progetto.

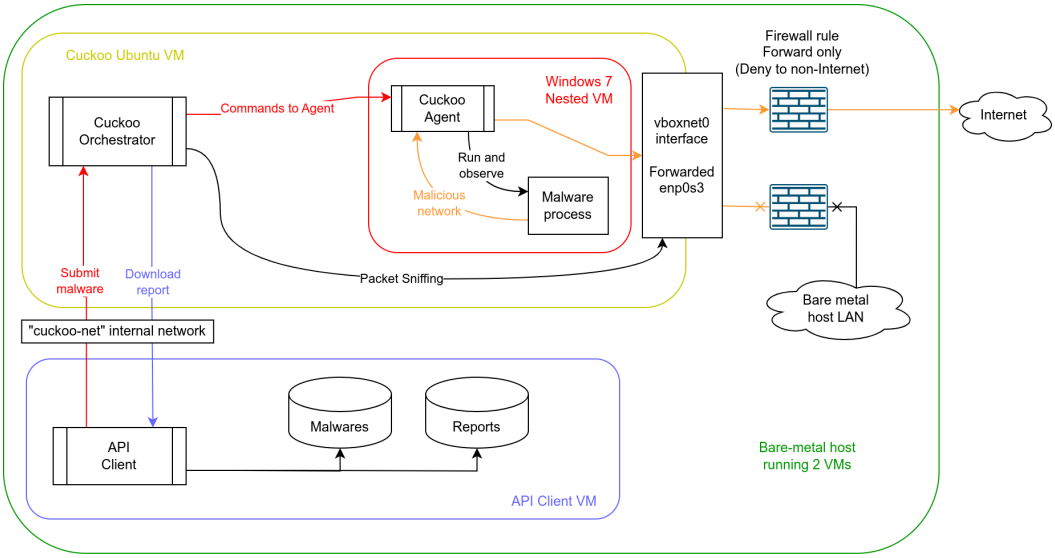


Figura 3.2. Architettura dell'infrastruttura di analisi dinamica

3.2.1 Installazione e configurazione

Una volta delineato lo schema di ciò che si vuole ottenere e il software alla base da cui partire, il primo passo è la sua installazione e configurazione. Non trattandosi di un normale software ma di una serie di componenti complessi da integrare, fin da subito la stessa installazione della sandbox si è dimostrata articolata.

Viene creata la macchina virtuale Cuckoo VM che, siccome conterrà al suo interno sia componenti complessi che altre macchine virtuali, deve avere allocate ingenti risorse. Si è scelto di riservare 8 core CPU e 8GB di RAM a tale scopo, con inizialmente solo 2 istanze delle VM innestate Windows 7. In caso si voglia scalare questo sistema su un server più potente, basterà solamente modificare le risorse allocate nella configurazione di VirtualBox in maniera semplice.

Il primo problema riscontrato è che, data la storicità di Cuckoo, è sviluppato in Python 2, mentre nelle più recenti versioni di Ubuntu la versione di Python di default è la 3, notoriamente non retrocompatibile. Viene così installato Python 2 manualmente, poi creato un virtual environment (`virtualenv`) per isolare le dipendenze del progetto dai pacchetti Python installati globalmente nel sistema e rendere il comando `python` di default un alias della versione 2, quando questo ambiente virtuale è attivato. Un virtualenv Python è definito dalla Python Enhancement Proposals PEP-405 [2] e fornisce gli strumenti per creare ambienti isolati tra loro, aventi ognuno le proprie dipendenze, versione dell'interprete e altre caratteristiche per non rompere il resto del sistema o progetti tra loro incompatibili, come in questo caso. Per impostare l'uso di una versione diversa da quella di default, si è usato il comando `virtualenv -python="$(which python2.7)" VENV_PATH/.venv`; l'attivazione si realizza con l'esecuzione di `source .venv/bin/activate` mentre ci si trova all'interno della directory del progetto.

Successivamente alle dipendenze Python, è stato richiesto di installare anche tutta una serie di servizi esterni. Il primo è chiaramente VirtualBox internamente alla VM stessa, per procedere con la virtualizzazione innestata. Poi sono venuti tutti i servizi necessari per il funzionamento dell'Orchestrator di Cuckoo, come il database PostgreSQL, MongoDB per la cache ed ElasticSearch per la correlazione dei dati. La scelta migliore per questi è stata la realizzazione di un file `docker-compose.yml` con tutti i servizi necessari e loro configurazioni (come variabili d'ambiente, mount di file o bind di porte di rete sull'interfaccia di loopback), installando perciò solo Docker e Docker Compose nel sistema, per avere massima compatibilità e minori problemi di setup.

Una volta che tutti i servizi accessori sono pronti per essere utilizzati, e dopo aver aggiornato i file di configurazione di Cuckoo per permetterne la raggiungibilità (impostando le variabili d'ambiente, le porte da contattare, etc), vanno attivati i servizi di Web interface e REST API. La prima ci servirà solamente in locale sulla VM stessa, non esposta all'esterno, a fini di debug e per un utilizzo iniziale più semplice. La seconda, al contrario, andrà esposta sulla rete interna *cuckoo-net* per permettere alla VM *API Client* di interagirvi. In questa fase di setup iniziale, viene tutto esposto su ogni interfaccia (bind su `0.0.0.0`) per facilitarne il debug, poi perfezionato durante l'Hardening (sez. 3.4).

[2] `https://peps.python.org/pep-0405/`

3.2.2 Creazione delle VM Windows

Per quanto concerne le VM innestate con Windows, si è scelto di adoperare la versione 7 del sistema operativo. Questa è ancora molto diffusa, soprattutto in ambito enterprise dove eventuali malware creerebbero maggiori danni, è supportata da Cuckoo allo stato attuale e, infine, ha opzioni per disattivare il maggior numero di funzionalità di sicurezza di Windows, quindi analizzare a pieno tutti i comportamenti che desidererà compiere il malware nell'ambiente di sandbox.

La creazione di una o più istanze del Guest è stata eseguita con `vmcloak`. Questo strumento si interfaccia con VirtualBox e consente la creazione di VM in maniera automatizzata e replicabile.

Viene assegnata una quantità di risorse paragonabile a un computer di fascia bassa verosimile. Il disco sarà di 128GB, allocato dinamicamente, con 4 core CPU e 2GB di RAM. Consentità un'esecuzione non eccessivamente limitante, senza sprecare troppe risorse, specialmente se si prevede di aumentare la parallelizzazione, come spiegato nella sez. 3.2.4.

Vengono poi installati al loro interno tutta una serie di strumenti di utilità che ci si aspetterebbe di trovare in un normale PC aziendale o personale. Questa caratteristica potrebbe sembrare superflua ma in realtà gioca un ruolo molto importante (assieme alle più avanzate tecniche che vedremo nella sezione 3.2.3):

- Serve come prima tecnica superficiale di anti-VM-detection, per cui un malware potrebbe rilevare se ci si trova in un ambiente di test o meno in base alla presenza o assenza di software che la quasi totalità degli utenti reali avrebbe sul proprio computer
- Ampia la superficie di attacco possibile sulla macchina Windows, in caso un malware non esegua direttamente il codice che genera danni nel sistema target ma, in maniera più subdola, vada a sfruttare falle di sicurezza presenti in altri programmi che comunemente un utente avrebbe
- Permette l'analisi di file diversi, come DOCX con macro non desiderate o PDF con payload eseguibili

Gli specifici software installati e operazioni effettuate sono:

- Adobe PDF Reader
- DotNet, Java, Flash e Visual C++ Redistributable 2015: per eseguire programmi che richiedono queste runtime
- Internet Explorer 11, Google Chrome e Firefox
- Modifica dello sfondo da quello di default a uno random
- WinRAR
- Windows 7 Service Pack 1
- Risoluzione dello schermo fissata a 1280x1024
- Suite Office

Chiaramente, è stata rigorosamente proibita l'installazione delle VirtualBox Guest Additions, che hanno lo scopo di integrare meglio la VM e fornire più servizi, ossia il contrario dell'obiettivo prefissato.

3.2.3 Anti-VM detection

Alcuni malware, per rendere più difficoltosa l'analisi in ambienti controllati, quindi rallentare lo sviluppo di un rilevamento accurato e una conseguente mitigazione, adottano tecniche di VM-detection. Se capiscono di essere sotto analisi, non eseguono affatto il comportamento che svolgerebbero su un ambiente reale, sia non compiendo azioni o compiendone di normali. Un esempio può essere un Trojan che, se sotto analisi, eseguirà regolarmente il programma autentico per cui si camuffa, compromettendo il sistema solo in ambienti reali.

Data la numerosità di queste tecniche, si è reso necessario trovare una PoC che possa eseguirle in maggior numero possibile e restituire un report su quali rilevazioni continuano a dare esito positivo. *Pafish64* è un progetto open source, rilasciato sotto licenza GPLv3 e disponibile su GitHub [3], che ha come scopo proprio ciò di cui necessitiamo in questo stadio. Procediamo ad eseguire il test sulle VM realizzate, senza ancora aver adottato alcuna strategia di mitigazione, per individuare dove e come operare. Il risultato, riportato in figura 3.3, è pessimo: un gran numero di test falliscono, tra cui:

1. tutti quelli che verificano la presenza di un potenziale essere umano, visualizzando un dialog con un tasto OK e aspettando che qualcuno lo vada a premere

2. la rilevazione dell'hypervisor VirtualBox che si sta utilizzando

3. alcuni controlli su aspetti più specifici della CPU

Come prima strategia, si vanno ad installare nelle VM le DLL fornite da Cuckoo per fare l'hooking delle chiamate di sistema che un processo eseguirà per tali rilevamenti, come nascondere alcuni registri di sistema intercettando le syscall `RegOpenKeyEx`, o eseguire OCR sullo schermo e muovere il mouse di conseguenza, compiendo azioni banali che un utente eseguirebbe con alta probabilità. Appena impostata la DLL injection, viene ripetuto il test. Il risultato è in figura 3.4 e presenta molti meno risultati negativi. Tuttavia, non si potrà progredire ulteriormente a causa dell'attuale architettura impostata e della ben maggiore difficoltà che avrebbe la realizzazione di soluzioni di mitigazione più sofisticate, al di fuori dello scopo del progetto qui realizzato. Infatti, a titolo esemplificativo, notiamo come l'hypervisor ottenuto leggendo i valori forniti dalla CPU (virtuale) sia *VBoxVBoxVBox*, riconducibile a VirtualBox; questo accade perché valori del genere vengono letti dalla macchina su cui la VM Windows 7 esegue, ma nella configurazione realizzata si tratta a sua volta di VirtualBox, per cui la strategia risulta inefficace.

Ci si ritiene però altamente soddisfatti del risultato poiché sono pochissimi i test che falliscono, abbassando la probabilità che un malware da analizzare utilizzi proprio quelli. Non può usarne in gran numero poiché sono rilevabili dalla sanbox

[3] `https://github.com/a0rtega/pafish`

tra le syscall eseguite, quindi riportate all'analista che può subito dedurre se si tratti di un comportamento lecito o inaspettato.

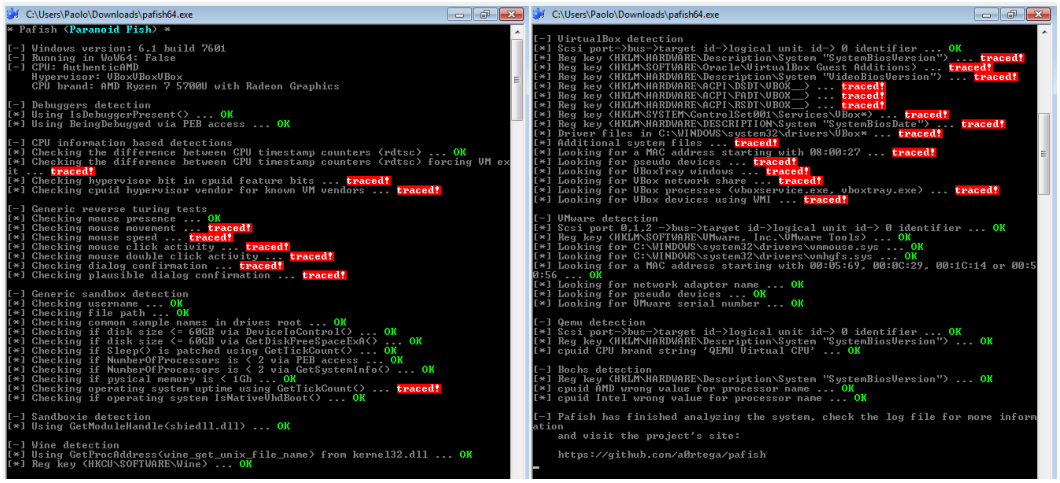


Figura 3.3. Pafish eseguito in una classica VM Windows 7

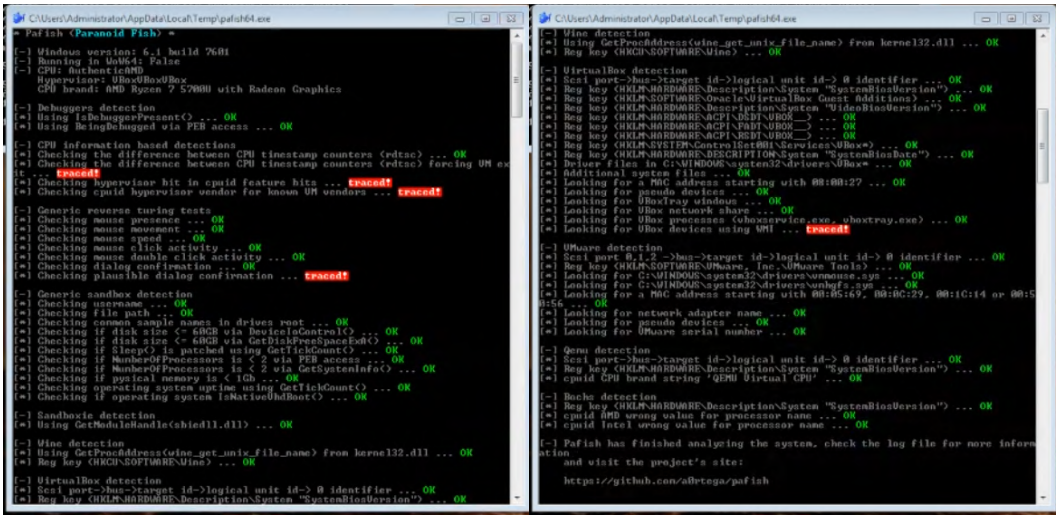


Figura 3.4. Pafish eseguito sotto l'analisi di Cuckoo Sandbox

3.2.4 Replicazione VM

Una volta realizzata una istanza della VM Windows 7, per migliorare la scalabilità del sistema, quindi permettere che più task possano eseguire in parallelo, si devono creare più istanze replicate della stessa.

Per deduplicare la base comune a tutte le istanze, come il sistema operativo e i tool installati, si trasforma il disco virtuale da standard a multi-attach, così da essere connesso a più VM in contemporanea.

A livello di rete si assegna un IP statico a ogni VM, consentendo a Cuckoo di correlare il traffico verso e da un determinato IP locale, a una specifica VM, quindi

a uno specifico task di analisi in corso in un determinato istante di tempo. Per permettere il forwarding dei pacchetti attraverso questa interfaccia, viene configurata usando il seguente comando a livello di orchestrator:

```
sudo sysctl -w net.ipv4.conf.vboxnet0.forwarding=1
sudo sysctl -w net.ipv4.conf.enp0s3.forwarding=1
```

Infine, si creano le istanze con vmcloak e si eseguono gli snapshot: questi verranno ripristinati subito dopo l'esecuzione di un task di analisi, così da ripartire sempre con lo stesso sistema pulito iniziale e non rischiare che un malware, dopo che infetti la VM, vada a mantenere il controllo della macchina virtuale anche successivamente.

3.2.5 Servizi systemd

Si nota come al boot, tutti i servizi che precedentemente erano funzionanti, debbano essere eseguiti manualmente. Questo passaggio è totalmente da rimuovere per una corretta automazione del processo, e la soluzione migliore in ambito Linux per eseguire servizi al boot è la creazione di *unit di systemd*. Systemd è un sistema di init e gestione dei servizi, ampiamente diffuso nella quasi totalità delle distribuzioni Linux, che deve eseguire con PID 1 (PID 0 è il processo scheduler, parte del kernel) per il corretto funzionamento. I servizi che deve avviare sono descritti attraverso dei file posti in `/etc/systemd/system/` chiamati **unit**. Come caratteristiche fondamentali, una unit conterrà:

- Un nome, dato dal file stesso, e una descrizione per permetterne il riconoscimento in futuro
- Uno o più servizi o target da cui dipende, che devono essere avviati prima di esso
- L'utente con il quale eseguire
- Il comando da lanciare per avviare ed interrompere il servizio
- Politiche di restart, che imposteremo ad *always* poiché necessitiamo che i vari componenti vengano riavviati automaticamente se dovessero smettere di funzionare

Vengono così create le seguenti unit:

- `vboxnet0.service` contiene il comando per creare e avviare preventivamente l'interfaccia di rete che useranno Cuckoo e le VM Guest per dialogare e far transitare il traffico di rete da sniffare
- `cuckoo-docker.service` avvia tutti i servizi accessori che Cuckoo necessita per il funzionamento, come i database, menzionati in 3.2.1, tramite un file Docker Compose
- `cuckoo.service`, che avvierà il demone di Cuckoo una volta che le dipendenze saranno pronte all'uso

- `cuckoo-api.service`, responsabile del server di REST API usato poi dal client esterno a questa VM
- `cuckoo-web.service` risulta utile quando, in fase di testing, si vuole usare una comoda UI per capire il funzionamento

```
[Unit]
Description=Cuckoo Web Service
Requires=cuckoo.service
After=cuckoo.service

[Service]
User=cuckoo
RemainAfterExit=yes
ExecStart=/home/cuckoo/cuckoo/bin/cuckoo web -H 127.0.0.1 -p 8000
TimeoutStartSec=0
Restart=always

[Install]
WantedBy=multi-user.target
```

Listing 9. Esempio di systemd unit per il servizio Web

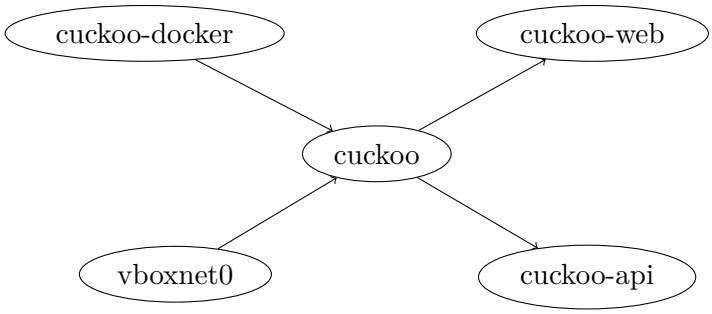


Figura 3.5. Grafo delle dipendenze dei servizi systemd aggiunti

3.3 VM Client con CLI

Seguendo l'architettura desiderata in figura 3.2, si procede alla realizzazione della API Client VM. Questa deve rispettare una serie di caratteristiche, come l'isolamento e la leggerezza, non contenendo particolari software.

Dovrà includere un programma client, scritto in Python, realizzato da zero per il progetto, utile sia come vero e proprio client che come Proof-of-Concept per una futura integrazione con le API della Threat Intelligence. Questo, infatti, interagisce direttamente con le REST API esposte da Cuckoo, esegue il parsing dei risultati e la modellazione delle chiamate in distinti metodi di una classe per

astrarne l'implementazione. Essendo già così suddiviso, rende possibile la scrittura di un altro programma Python per essere usato in un diverso ambiente, riciclando il codice relativo alle chiamate REST. Ciò porta a un risparmio di risorse umane per futuri sviluppi e alla minor probabilità di introdurre bug, riutilizzando codice già testato in precedenza.

Il Client può essere usato come libreria o essere eseguito direttamente da terminale come modulo (`python3 -m cuckoo-api-client ARGS...`), passando diverse possibili opzioni, tra cui l'analisi di un file o di un'intera cartella, e l'impostazione di un path dove andare a salvare tutti gli output. Questi ultimi consistono nel report JSON elaborato, nonché gli screenshot dell'esecuzione e il file pcap con la cattura dei pacchetti di rete. Per agevolarne l'usabilità, fornisce messaggi di errore e di help tramite l'ausilio del modulo `argparse`, e durante l'esecuzione mostra output colorato e in costante aggiornamento sullo stato dei task, che siano in coda, terminati o pendenti.

La VM ne conterrà una copia già al suo interno, così da ridurre dipendenze esterne o necessità di mount di cartelle dall'host fisico. Per la massima leggerezza, è basata su Arch Linux con installate solo le dipendenze strettamente necessarie per il funzionamento, e XFCE come Desktop Environment minimale per la visualizzazione di screenshot o altri file che possono essere più comodamente aperti utilizzando programmi con una GUI. Grazie a questa scelta, è stato possibile mantenere la dimensione della macchina virtuale, esportata come file OVA, inferiore a 3GB.

Avrà un indirizzo IP statico per facilitarne la raggiungibilità e sarà connessa all'interfaccia di rete interna *"cuckoo-net"*, come descritto nell'architettura.

3.3.1 Nuovo formato del report

Dopo una prima osservazione del report originale, così come elaborato da Cuckoo, ci si è resi conto della sua bassa consultabilità, risultando eccessivamente lungo e dispersivo. Inoltre, alcune voci non sono ben organizzate, come le syscall intercettate che non presentano una chiara distinzione tra i vari thread del processo ma sono elencate tutte assieme. Questo comporta che, in caso di due o più thread che eseguono operazioni distinte, non è facile ricostruire il flusso logico semplicemente leggendo questo elenco, poiché i vari flussi sarebbero uniti. Per questa e altre ragioni di comodità da parte dell'analista, oltre a motivazioni più tecniche dovute alla struttura del JSON con cui generalmente lavora la Threat Intelligence, si è optato per eseguire una trasformazione lato client.

Per prima cosa, viene studiata la struttura del JSON fornito da Cuckoo per capire come e dove intervenire. Nel primo livello, si tratta di un oggetto (e non una lista), con diverse chiavi. Questa caratteristica risulta utile: restituire come dato un oggetto rende successivi aggiornamenti o aggiunte alla struttura dati molto più agevole, rispetto a una lista o altre strutture più basilari. Le prime chiavi che sono ben strutturate e andremo a ricopiare nel report trasformato sono i metadati (`target` e `task_info`), i dati sugli screenshot e le stringhe individuate nel file. Le sezioni dov'è richiesto il maggior intervento per organizzare meglio, secondo le esigenze, sono quella relativa ai processi e syscall, e l'area di network con le analisi del traffico di rete.

Le informazioni sui processi vengono fornite in maniera estremamente ridondante e dispersiva: vengono elencati tutti i processi, senza una struttura ad albero, e le syscall relative non sono organizzate nei vari thread, infine sono ripetute varie volte in diverse sottosezioni. Il nuovo report conterrà la seguente struttura:

- Una foresta dei processi, dove ognuno ha i propri figli direttamente nella struttura JSON: si parla di foresta e non di albero perché, nel sottoinsieme dei processi analizzati, possono esistere più radici, quindi più alberi distinti tra loro
- Per ogni processo, è identificato il PID, il path da cui è stato lanciato, i match con YARA rules identificati, assieme a tutta una serie di statistiche sul numero di syscall eseguite per categoria - come *storage* o *network*
- Un processo a sua volta avrà un insieme di thread, ciascuno col proprio elenco di syscall, ordinate cronologicamente per istanze di invocazione
- Ogni chiamata a sistema conterrà informazioni quali il nome mnemonico, la categoria, i parametri in input e il valore restituito in output

La sezione riguardante il traffico di rete, vede una divisione (prima non presente) tra le categorie di traffico e protocollo usato, come HTTP, HTTPS, TCP con protocollo applicazione custom, SMTP per invio di mail, richieste DNS, UDP o altri protocolli supportati. La divisione, inoltre, serve a far ignorare all'analista tutto ciò che non è rilevante, focalizzandosi su elementi anomali, come una calcolatrice che invia traffico SMTP verso un server con IP non conosciuto. Per tutti i protocolli, sono riportati gli indirizzi IP sorgente e destinazione, da cui possiamo dedurre sia l'interlocutore che la direzione della comunicazione, così come le porte coinvolte e il timestamp di esecuzione. Protocolli di livello applicazione, come HTTP, contengono informazioni già elaborate. Proprio in questo caso d'esempio, viene restituita una struttura dati ad oggetto dove sono già presenti in maniera ordinata i vari header, lo status code e il body di richiesta (se presente) o risposta. Se possibile, vengono decrittate le comunicazioni TLS, sia con metodologie MITM (man-in-the-middle, sconsigliato e rilevabile dal processo con tecniche quali il Certificate Pinning) che leggendo dai buffer che vengono inviati alle syscall durante queste operazioni, a patto che non abbia linking statico alle librerie OpenSSL e simili, ma usi le syscall fornite dal sistema operativo e poi intercettate da Cuckoo Sandbox.

Una caratteristica che va tenuta a mente quando si analizzano i risultati della parte di networking è che si basano principalmente sul traffico catturato ascoltando l'interfaccia di rete. Come conseguenza di ciò, emerge come nel risultato siano presenti anche pacchetti che il processo sotto analisi non ha mai inviato o ricevuto, ma attribuibili ad altri processi all'interno del sistema, come software di Microsoft che eseguono operazioni di routine (ricerca di update del sistema operativo, etc...). Da un lato può essere visto come un punto a favore, così che se per qualche vulnerabilità di programmi di sistema, venisse effettuato traffico di rete malevolo partendo proprio da questi, sarebbe comunque rilevato e analizzato dall'analista. In maniera speculare però è un punto a sfavore l'aver il risultato dell'analisi meno pulito e con elementi superflui. Si è scelto, allo stato attuale, di rimandare tale decisione all'analista e al sistema di Threat Intelligence poi, ad esempio eseguendo una analisi di un

programma mockup che non esegua alcuna operazione di rete, per estrarre così il traffico che ci possiamo aspettare da altre future analisi. Ai fini del progetto, vengono conservati tutti i pacchetti.

```
{
  "process_forest": [{
      "pid": 1548,
      "is_root": true,
      "children": [{
          "pid": 2128,
          "is_root": false,
          "children": []
        }]
    }],
  "processes": {
    "2128": {
      "process_name": "win-http.exe",
      "summary": {
        "connects_ip": ["93.184.216.34"],
        "resolves_host": ["example.com"]
      },
      "signatures": [],
      "apistats": {
        "getaddrinfo": 1,
        "socket": 1
      },
      "modules": [{
          "baseaddr": "0x77110000",
          "filepath": "C:\\Windows\\SYSTEM32\\ntdll.dll"
      }],
      "main_thread_id": 884,
      "thread_calls": {
        "884": [{
            "category": "network",
            "status": 1,
            "api": "socket",
            "return_value": 96,
            "arguments": {
              "protocol": 0,
              "socket": 96,
              "af": 2,
              "type": 1
            },
            "tid": 884
        }]
      }
}}}
```


Listing 10. Esempio di output per le informazioni sui processi, sintetizzato

```
{
  "network": {
    "tls": [],
    "udp": [
      {
        "src": "192.168.56.101",
        "dst": "20.101.57.9",
        "offset": 515017,
        "time": 6.215277194976807,
        "dport": 123,
        "sport": 123
      }
    ],
    "dns_servers": [
      "8.8.8.8"
    ],
    "icmp": [],
    "tcp": [
      {
        "src": "192.168.56.101",
        "dst": "51.132.193.105",
        "offset": 522251,
        "time": 11.470929145812988,
        "dport": 443,
        "sport": 49165
      }
    ],
    "dns": [
      {
        "type": "A",
        "request": "example.com",
        "answers": [
          {
            "data": "93.184.216.34",
            "type": "A"
          }
        ]
      }
    ],
    "http": [
      {
        "status": 304,
        "src": "192.168.56.101",
        "protocol": "http",
        "dst": "93.184.221.240",
        "uri": "/msdownload/update/v3/static/trustedr/en/authrootstl.cab",
        "host": "www.download.windowsupdate.com",
```

```
        "dport": 80,
        "sport": 49191,
        "method": "GET",
        "request": {
          "headers": "Connection: Keep-Alive\r\nAccept: */*",
          "content_sha1": "da39a3ee5e6b4b0d3255bfef95601890afd80709"
        },
        "response": {
          "headers": "HTTP/1.1 304 Not Modified\r\nAccept-Ranges: bytes",
          "content_sha1": "da39a3ee5e6b4b0d3255bfef95601890afd80709"
        }
      },
      {
        "status": 200,
        "src": "192.168.56.101",
        "protocol": "http",
        "dst": "93.184.216.34",
        "uri": "/",
        "host": "example.com",
        "dport": 80,
        "sport": 49171,
        "method": "GET",
        "request": {
          "headers": "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close",
          "content_sha1": "da39a3ee5e6b4b0d3255bfef95601890afd80709"
        },
        "response": {
          "headers": "HTTP/1.1 200 OK\r\nCache-Control: max-age=604800",
          "content_sha1": "4a3ce8ee11e091dd7923f4d8c6e5b5e41ec7c047"
        }
      }
    ]
  }
}
```


Listing 11. Risultato trasformato per la parte di networking, inclusi i pacchetti provenienti da altri processi

3.4 Hardening

Un sistema di sandbox è molto delicato e deve essere protetto adeguatamente. La VM dove eseguono Cuckoo e i guest innestati rappresentano il punto critico, dato che una loro errata configurazione potrebbe aprire le porte all'attaccante verso l'intero sistema host ed evadere così dalla sandbox. Adotteremo il principio del minimo privilegio (Principle of Least Privilege - PoLP).

A livello di rete, si è notato come un processo interno alla sandbox (Windows 7) possa riuscire a contattare con successo altri dispositivi nella LAN del sistema host fisico. Questo deve essere chiaramente proibito, poiché un malware potrebbe scoprire e attaccare altri dispositivi vulnerabili nella propria rete domestica o aziendale. Per far ciò, vengono impostate delle regole precise di IPTables. Useremo il programma `ufw` per applicare le regole, dato che sotto la scocca usa comunque IPTables ma con un'interfaccia per l'utente molto più intuitiva e meno error-prone, restando pur sempre su CLI. Blocchiamo, dalla VM cuckoo più esterna, tutto il traffico verso l'host (interfaccia `enp0s3`) che sia indirizzato a classi di IP privati. Nell'esempio di seguito, applichiamo la regola per proibire di raggiungere tutti i dispositivi in una rete con indirizzi privati di classe C; verrà replicato per tutte le altre classi di indirizzi da bloccare.

```
sudo ufw deny out on enp0s3 to 192.168.0.0/16
sudo ufw enable && sudo ufw reload
```

Per quanto riguarda il gateway, dobbiamo consentire il traffico che transita verso di esso, ma non direttamente a lui destinato. La regola è applicabile con il seguente comando di ufw, dove `10.0.2.2` è l'indirizzo IP del gateway, concretizzato nell'host fisico dove gira VirtualBox:

```
sudo ufw deny out on enp0s3 to 10.0.2.2
```

La comunicazione tra la VM di analisi e la VM client, che transita sull'interfaccia di rete interna di VirtualBox denominata `cuckoo-net`, poi assegnata al sistema con nome `enp0s8`, deve essere controllata. Ricordando che le due macchine hanno assegnati i propri IP statici, dobbiamo limitare il traffico facendo sì che solo la VM client sia in grado di aprire una connessione verso l'analizzatore, limitatamente alla porta 8080. Oltre a limitare sul client tutte le connessioni in ingresso a livello di firewall, anche se esista una porta aperta con bind su 0.0.0.0 (tutte le interfacce), va applicata la seguente regola sull'analizzatore, tenendo `deny` come azione di default per il traffico in ingresso:

```
sudo ufw allow in \
    on enp0s8 \
    to 192.168.50.11 \
    port 8080 proto tcp \
    from 192.168.50.12
```

questa permette comunicazioni solo sulla porta 8080 dove è in ascolto il server REST API di Cuckoo, tramite una connessione TCP, dall'interfaccia di rete condivisa e isolata tra le due macchine, unicamente dall'indirizzo IP statico a lei assegnato.

Oltre alla rete, si vuole proteggere anche tutto l'ambiente di esecuzione interno all'analizzatore, quindi si è creato un utente Unix distinto, denominato `cuckoo`, da cui lanciamo VirtualBox e non dispone di alcun permesso, tantomeno è nel gruppo sudo (i cosiddetti *sudoers*). La sua password è bloccata e la shell è impostata a `/usr/bin/nologin` così da inibirne completamente l'accesso. Per l'avvio però è presente un utente `pc` che può eseguire il programma di VirtualBox e aprire una shell impersonando l'utente meno privilegiato utilizzando sudo:

```
xhost +SI:localuser:cuckoo
sudo -u cuckoo virtualbox
```

Il comando di `xhost` si è reso necessario per far sì che anche la GUI del programma possa essere correttamente visualizzata. XHost è una utility del server grafico X, e con questo comando si autorizza l'utente meno privilegiato a contattarlo per mostrare finestre o altri elementi visivi nella sessione dell'utente corrente.

3.5 Plugin aggiuntivi

Si è optato per estendere la sandbox, integrando altri software più innovativi e fornire una metodologia standard per un'integrazione di ulteriori strumenti in futuro. I tool in questione richiesti sono:

- **Patriot**, [4] piccolo progetto di ricerca per identificare tecniche di occultamento in memoria (*in-memory stealth*) usando in maniera non standard alcune chiamate di sistema di Windows
- **Hunt-Sleeping-Beacons**, [5] che ha lo scopo di identificare i beacons decompressi a runtime, o in esecuzione nel contesto di altri processi; si tratta di agenti C&C (Command and Control) e si osservano beacon che attendono tra loro, andando in sleep e causando ritardo nell'esecuzione, evitando di essere intercettati in una normale analisi tradizionale

Approfondendo quest'ultimo strumento, le tecniche di sleep obfuscation che va a rilevare includono casistiche come *DelayExecution*, in cui un processo agent C2 andrà a richiamare `Kernel32!Sleep`, che però indirettamente invoca `Ntdll!NtDelayExecution`, portando il processo in stato di waiting; ma anche tecniche più avanzate come *Foliage*, che usano una serie di syscall per ritardare l'esecuzione senza tuttavia transitare per lo stato di DelayExecution, risultando molto meno rilevabili in modi convenzionali, fino a rilevare *Ekko* o implementazioni analoghe ancora più sofisticate.

3.5.1 Download durante l'analisi

Entrambi questi tool vengono forniti dai rispettivi autori come eseguibili precompilati o facilmente compilabili dai sorgenti pubblici con MinGW. Questi devono tuttavia essere eseguiti durante l'analisi all'interno della VM guest Windows 7. Per fare ciò, non è stato identificato alcun metodo già fornito da Cuckoo Sandbox, allora è stato

[4] `https://github.com/joe-desimone/patriot`
[5] `https://github.com/thefLink/Hunt-Sleeping-Beacons`

realizzato un piccolo server HTTP in Python che, eseguendo sulla VM Ubuntu ed esposto su una porta definita (50001) limitatamente all'interfaccia di rete nominata *enp0s8* condivisa tra le 2 VM VirtualBox.

Per ridurre la superficie di attacco, non si va ad usare il modulo `http.server` già presente nella libreria standard poiché potrebbe essere vulnerabile ad attacchi Path Traversal o analoghi. Come soluzione adottata, si crea un semplice server HTTP che accetta solo richieste GET e, a seconda dell'endpoint, viene scelto il percorso del file da una serie di stringhe hardcoded, così da rendere impossibile Path Traversal in quanto l'input utente non è mai utilizzato per accedere al file, ma viene solo confrontato con una serie di stringhe.

Al contrario, la comunicazione nella direzione inversa, quindi dal guest all'host, è gestita tramite la API di Cuckoo attraverso la funzione `upload_to_host`, dove si può passare il path di un file da copiare e successivamente da esaminare.

3.5.2 Esecuzione come plugin Cuckoo

Successivamente all'installazione dei tool nell'ambiente di sandbox, è necessario eseguirli e integrarli all'interno del flusso di analisi di Cuckoo. Questo è possibile grazie all'implementazione di plugin in Python 2 usando le API della sandbox.

Si compone di due distinte componenti, entrambe da realizzare in Python 2:

- modulo **Auxiliary**: esegue nella VM Guest e al termine deve riportare i dati ottenuti alla VM Host per l'analisi
- modulo **Processing**: esegue nella VM Host e riceve i dati di output forniti dall'auxiliary

Partendo dai moduli auxiliary, molto simili tra loro, lo scopo è scaricare ed eseguire il programma corrispondente del modulo, ripeterlo in loop per collezionare il maggior numero possibile di match, infine organizzare i report eliminando i duplicati e riportarlo all'host. Questo tipo di moduli è concretizzato in un file Python con una classe custom che estende la classe astratta `Auxiliary` e implementa i metodi `start` con cui avviare l'esecuzione e segnala che è il file da analizzare sta per essere avviato, così da dare modo al plugin di predisporre tutti gli strumenti necessari, che siano tool, hook, capture di rete, o altro; e il metodo `stop` necessario per interrompere e ripulire l'ambiente d'esecuzione. Inoltre, siccome il loop di esecuzione continua del tool deve essere non bloccante, altrimenti l'intero flusso di esecuzione dell'agent di Cuckoo viene bloccato, senza di fatto eseguire mai, deve essere delegato in un thread in background. Sarà presente un flag che, impostato inizialmente a True prima dell'esecuzione, verrà settato a False durante lo stop del modulo, così da interrompere il thread con una graceful exit. Lo standard output (stdout) è conservato dentro un file di log, poi inviato al modulo di Reporting.

3.5.3 Parsing dei risultati

La parte di analisi del log raccolto è svolta nei moduli di Reporting che, come annunciato, sono eseguiti all'interno della VM host da parte dell'orchestrator. Sono delle classi che estendono la classe astratta `Reporting` e devono implementare il metodo `run()` che restituirà al chiamante la struttura da inserire nel report finale.

Ricevono i dati caricati dall'auxiliary e verranno trasformati in apposite strutture, codificate in formato JSON, restituite nel report finale complessivo poi inviato al client che ha richiesto l'analisi. Il modo in cui vengono interpretati i risultati è fortemente dipendente dal tool e dal suo output specifico, per cui in questo caso i moduli mostreranno sostanziali differenze date dalla natura dello strumento stesso. Di seguito ne analizziamo il comportamento.

Patriot

L'output testuale è dato in input al parser nel formato mostrato in figura 3.6. Si evince facilmente come i dati di nostro interesse siano già mostrati come chiave-valore, dove le chiavi sono gli elementi `Level`, `Type`, `Detail`, `PID` e `Process`. Sarà sufficiente notare come le varie sezioni sono divise da una serie di trattini e una linea vuota, ossia la sequenza di caratteri `\n\n`. Per ogni riga di ogni sezione, basta dividere tra chiave e valore andando a fare uno split della stringa alla prima occorrenza del carattere : e unendo il tutto dentro un dizionario, poi restituito nell'output trasformato. Possono essere rimossi tutti i duplicati, dovuti all'esecuzione in loop dello strumento, come spiegato nella sezione precedente.

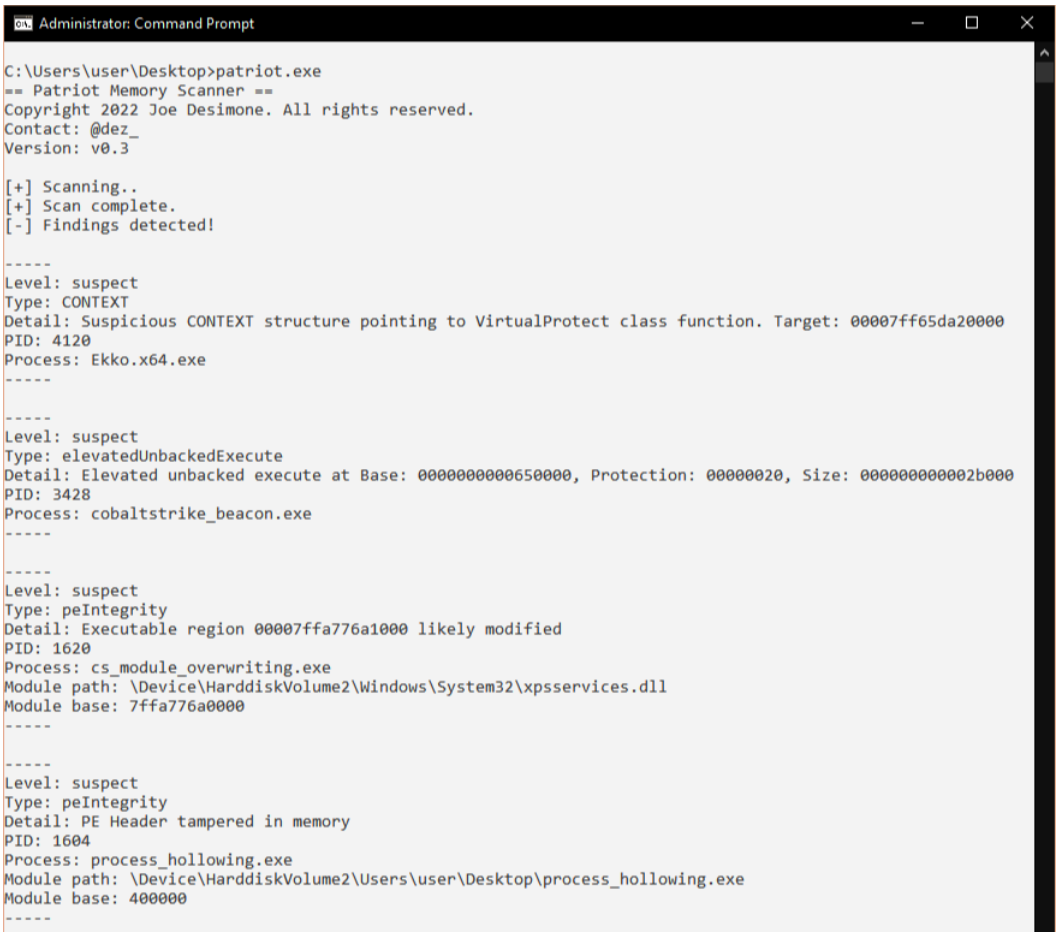


Figura 3.6. Esempio di output testuale di Patriot

```
[
    {
        "Level": "suspect",
        "Type": "peIntegrity",
        "Detail": "Executable region 000007fefd131000 likely modified",
        "PID": 2656,
        "Process": "SDXHelper.exe",
        "Module path": "\\Windows\\System32\\KernelBase.dll",
        "Module base": "7fefd130000"
    },
    {
        "Level": "suspect",
        "Type": "peIntegrity",
        "Detail": "Executable region 000007feff211000 likely modified",
        "PID": 2656,
        "Process": "SDXHelper.exe",
        "Module path": "\\Windows\\System32\\oleaut32.dll",
        "Module base": "7feff210000"
    }
]
```


Listing 12. Esempio di output di Patriot trasformato in JSON dal plugin

Hunt-Sleeping-Beacons

Nel caso specifico di questo strumento, siccome l'output testuale che fornisce è in un formato molto flessibile e può essere espresso in una moltitudine di modi diversi, si è scelto di non trasformarlo in una struttura a dizionario. Si è ritenuto notevolmente più utile, ai fini della rilevazione, filtrare unicamente le righe inizianti per `[!]` le quali indicano la motivazione e il target (nome del processo e PID) di una detection. Le righe così ottenute possono essere filtrate eliminando i duplicati e restituendole in output come una lista.

```
[!] Suspicious Process: beacon.exe (5296)

   [*] Thread (2968) State: DelayExecution and uses stomped module
   [*] Potentially stomped module: C:\Windows\SYSTEM32\xpsservices.dll

       NtDelayExecution -> C:\Windows\SYSTEM32\ntdll.dll
       SleepEx -> C:\Windows\System32\KERNELBASE.dll
       DllGetClassObject -> C:\Windows\SYSTEM32\xpsservices.dll

   [*] Suspicious Sleep() found
   [*] Sleep Time: 5s
```

Listing 13. Esempio di output per una detection di Hunt Sleeping Beacons

3.6 Workflow analisi dinamica

Successivamente alla creazione dei singoli elementi, riassumiamo l'intero workflow finale di un processo di analisi dinamica.

In primo luogo, sarà richiesto avviare la VM denominata *Ubuntu-cuckoo* in background, senza necessitare dell'uso della UI o di altre interazioni, se non in caso di debugging. Grazie ai servizi systemd creati, è stato reso automatico l'intero avvio, così da non richiedere alcun intervento manuale.

Stando all'architettura indicata a inizio capitolo in figura 3.2, i malware samples risiedono nella VM *API client*. Per l'attuale configurazione, vengono salvati o direttamente nella VM o in locale sull'host, in una cartella montata nella macchina virtuale in sola lettura; ricordando i rischi identificati nella sezione relativa all'hardening, si è valutato questo come il miglior compromesso tra sicurezza e comodità d'utilizzo, rispetto ad altre soluzioni come connettere questa VM direttamente ad Internet o all'host via servizi di rete.

Avviate entrambe le macchine virtuali costruite, si va ad interagire con la macchina client avviando un terminale ed eseguendo il programma Python client sviluppato su misura (sez. 3.3). Prenderà come argomenti il file o la cartella con uno o più sample da analizzare, nonché la cartella dove salvare gli output. Per ridurre il più possibile l'utilizzo dell'input utente nelle fasi di analisi, si è deciso di salvare l'output in una sottocartella avente per nome l'impronta SHA256 del file, e non il filename originale. Infatti, in caso di errori in una futura automazione, nell'evenienza di una vulnerabilità del filesystem, usare il nome del file così come ricevuto potrebbe allargare la superficie di attacco della sandbox. Inoltre, presenta il grande vantaggio di raggruppare nella stessa directory i risultati di più analisi relative allo stesso file, seppur ricevuto con nomi distinti, ad esempio *malware.exe* e *malware (1).exe*.

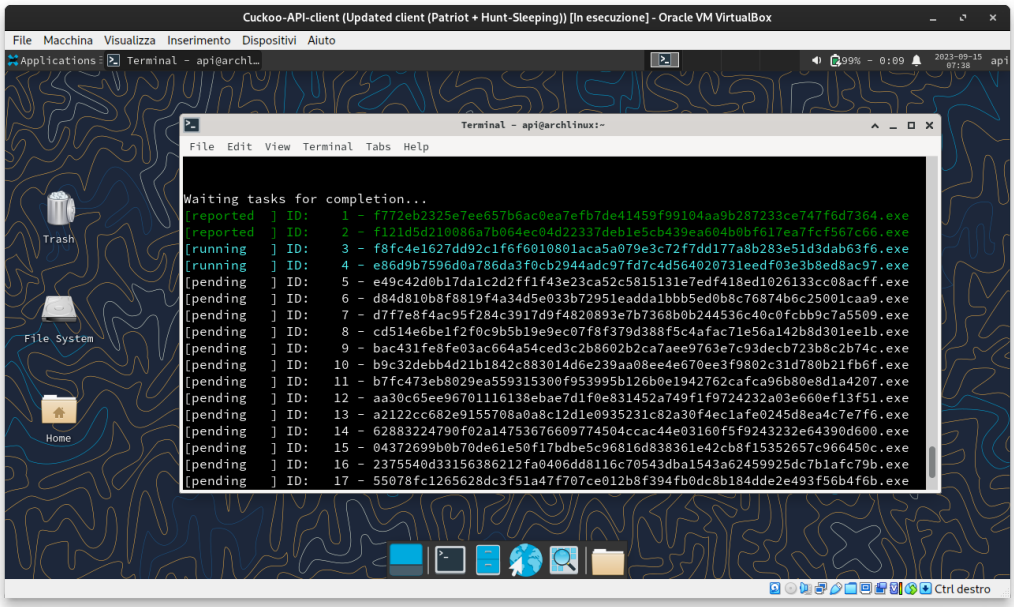


Figura 3.7. Output del client Python durante l'esecuzione dei task

I risultati finali, contenuti in questa cartella, saranno di 3 tipi:

- Il report JSON complessivo del behaviour, già trasformato nel nuovo formato descritto nella sez. 3.3.1
- Gli screenshot scattati durante l'esecuzione del sample, in caso mostri degli elementi grafici
- Il file PCAP di cattura di rete contenenti tutti i pacchetti transitati sull'interfaccia che collega la VM a Internet, tramite l'host fisico che opera da gateway

Al fine di distinguere risultati relativi allo stesso sample ma eseguiti in momenti differenti, i file avranno come prefisso il timestamp di avvio del task.

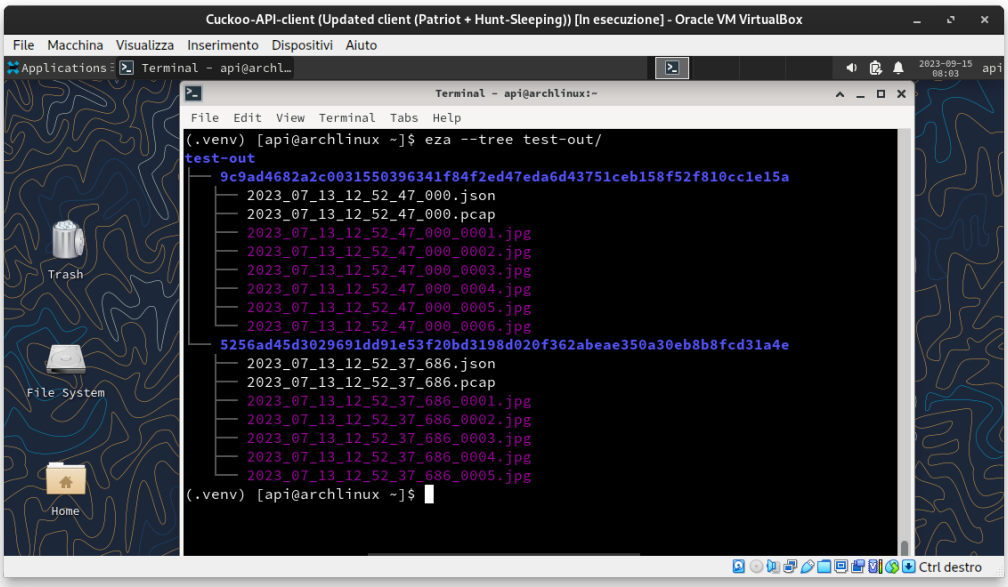


Figura 3.8. File dei risultati per i vari sample

Capitolo 4

Sviluppi futuri

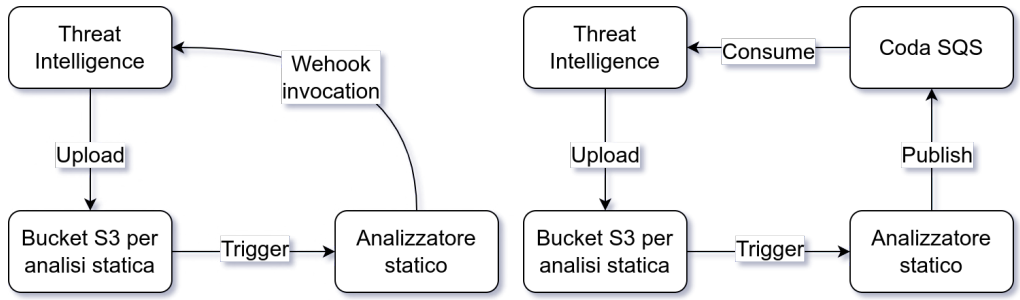
Il progetto qui realizzato rappresenta la base su cui poggerà l'infrastruttura aziendale descritta originariamente nella figura 1.3. Verranno quindi sostituite le attuali integrazioni con servizi mockup o di terze parti di test con le reali componenti aziendali, ancora in fase di sviluppo internamente.

4.1 Integrazione con Threat intelligence

Il primo punto cruciale con cui si lega a doppio filo è il sistema di Threat Intelligence automatico (di seguito *"T.I."*). Questo, successivamente alla propria ultimazione, si integrerà agli analizzatori statici e dinamici. Per meglio specificare, lo statico è già predisposto: sarà sufficiente effettuare l'upload sul bucket S3 designato per avviare il processo. Inoltre, per ricevere i risultati, la T.I. esporrà un endpoint HTTP dove ricevere notifiche riguardanti l'avvio, la terminazione o eventuali errori di un task relativo a un sample caricato. Questo sistema di notifiche segue il paradigma dei webhook, ossia un sistema di callback implementato a livello di richieste HTTP. In questo modo, già predisposto, il richiedente non dovrà attendere anche svariati minuti con la connessione TCP aperta finché il server non abbia terminato la propria elaborazione, con il rischio di perdere tutto in caso di interruzioni di rete. Al contrario, gestirà l'avanzamento tramite endpoint dedicati, senza dover allocare risorse durante l'esecuzione. Questa integrazione è resa dal progetto estremamente facile, perché le librerie per eseguire richieste HTTP sono già incluse, assieme alla definizione dei tipi di dati scambiati. Siccome questa modifica sarà obbligata, nel codice sono stati inseriti dei commenti *TODO* esattamente nei punti specifici dove cambiare tale comportamento, al fine di rendere la loro identificazione molto semplice per lo sviluppatore che lavorerà con questo progetto in futuro.

Un sistema più avanzato potrebbe prevedere l'uso di code FIFO per i messaggi. AWS implementa tale funzionalità attraverso il proprio prodotto denominato **SQS** (*Simple Queue Service*), dove l'analizzatore avrà il ruolo di producer e il richiedente sarà il consumer. Così si aumenta la *fault tolerance*, essendo che i messaggi di output resteranno nella coda, lasciando tutto il tempo al servizio della T.I. di tornare in funzione, senza perdere dati.

Passando a integrare anche l'analizzatore dinamico, ciò comporterebbe il piccolo sviluppo di una API, possibilmente REST, che consenta di invocare l'analisi per



(a) Risposta mediante webhook **(b)** Risposta mediante uso di coda SQS

Figura 4.1. Potenziali integrazioni tra la T.I. e l'analizzatore statico

dei sample specifici. Come accennato in precedenza, non deve essere eseguito un task in concomitanza alla parte statica. Questo perchè deve essere compito della Threat Intelligence, o dell'analista in persona, decidere se ha senso spendere risorse nell'analisi dinamica o meno, sulla base dei risultati della prima o l'uso di una threshold di confidence determinata secondo i più svariati criteri.

La soluzione per cui il progetto è stato predisposto è la seguente:

- Viene realizzata una REST API, che espone un endpoint per l'avvio di un task di analisi dato il sample, e che dialoga con Cuckoo utilizzando come base il modulo Python costruito nella sezione 3.3: per via della libreria già pronta e testata, lo sforzo umano richiesto si prevede sia notevolmente basso
- La T.I. andrà ad esporre un suo webhook per la ricezione della risposta o degli errori provenienti dalla Sandbox
- La VM con l'analizzatore, al termine del task, dove ora semplicemente salva i risultati negli appositi file e mostra l'output di successo all'utente, andrà a invocare il webhook di callback inviando i risultati; in alternativa potrebbero essere caricati su un bucket S3 a fini di archiviazione

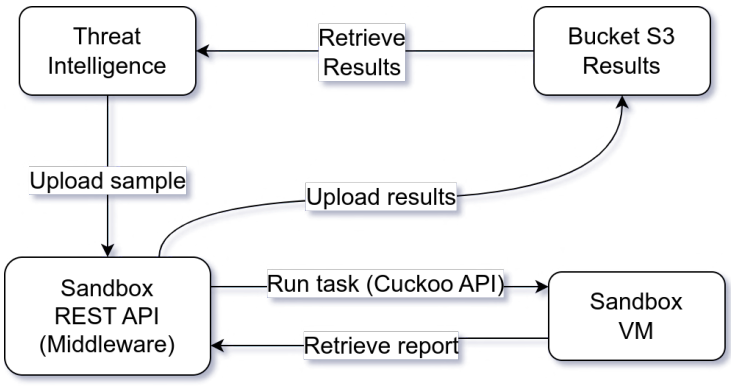


Figura 4.2. Possibile integrazione futura della sandbox con la Threat Intelligence

Com'è possibile notare, si tratta di operazioni relativamente semplici, che non richiedono un grande quantitativo di forza lavoro. Questo è il risultato positivo

derivante dall'architettura del progetto, mantenendo il focus sempre sulla scalabilità e adattabilità dello strumento. Tuttavia, la parte più dispendiosa, potrebbe riguardare l'acquisto e l'impostazione iniziale di un server fisico reale predisposto per l'esecuzione della sandbox.

4.2 VPN

Un fattore che potrebbe permettere al malware di capire che si trova sotto analisi da parte di un'azienda di sicurezza, è l'indirizzo IP pubblico col quale dialoga su Internet. Nel caso di un agent C&C (command and control), il malintenzionato vedrebbe comparire nel proprio pool di computer infettati anche una connessione avente per capo un IP pubblico. Potrebbe andare ad eseguire un reverse DNS su tale indirizzo, scoprendo facilmente che appartiene a un'azienda di cybersecurity. L'effetto finale è quello che l'attaccante non andrà presumibilmente a far eseguire a quel computer (di fatto la nostra VM con Windows 7) payload malevoli, ma bensì potrebbe ordinare l'autodistruzione e andare ad applicare strategie di mitigazione, come il cambio dell'host centro di comando.

Una soluzione efficace è quella di inserire la sandbox all'interno di una VPN retail, aiutata dal loro sempre maggiore utilizzo da parte degli utenti comuni. Così facendo, le richieste verso Internet verrebbero viste dall'attaccante come provenienti da un servizio VPN usato da milioni di utenti per il mondo, nascondendo il fatto che si tratti di un'azienda specializzata nel settore.

Figura 4.3. Transito dei pacchetti di rete dalla sandbox alla VPN verso Internet

4.3 Deploy totale su AWS

Un passaggio che si è valutato durante l'esecuzione del progetto è stato quello di far eseguire le macchine virtuali direttamente su cloud attraverso servizi AWS, al posto di adoperare un server fisico aziendale. Ciò minimizzerebbe i rischi di evasion e infezione del resto dell'infrastruttura aziendale locale, ma non va dimenticato l'aspetto relativo alla fattibilità tecnica e legale. Bisogna tenere conto che è richiesto che l'host fisico su cui esegue la sandbox supporti sia la virtualizzazione standard che innestata. Quest'ultima non è molto comune tra i server AWS poiché non se ne vedono applicazioni più generiche. Infine, sarà una migrazione valutabile solo a posteriori di una consulenza legale per assicurarsi di non violare i termini di utilizzo di AWS, nello specifico dei prodotti di EC2 (per il server), nonché dell'infrastruttura di rete su cui transiterebbe indisturbato del traffico potenzialmente malevolo o facente parte di una botnet.

4.4 Uso di Cuckoo Sandbox su Python 3

È stato sottolineato come sia stato utilizzato il software Cuckoo Sandbox nella versione originale, realizzata dalla Cuckoo Foundation, scritta in Python 2. Oltre a dipendere da una versione dell'interprete ormai deprecata e da librerie obsolete, sia perché non più mantenute che non più riceventi aggiornamenti di sicurezza, anche il progetto stesso è stato archiviato e non vedrà migliorie future, come chiaramente esplicitato sulla propria pagina GitHub. [1]

Tutto ciò è dovuto dalla riscrittura della nuova versione basata su Python 3, che inevitabilmente romperà la retrocompatibilità con l'attuale. Non è stata usata in questo progetto perché ancora non disponibile al pubblico, ma è programmato il suo utilizzo in una fase futura, dopo che sarà dichiarata stabile e testata a sufficienza. Inoltre, per l'uso per il quale verrà sfruttata, l'azienda richiederà che vengano prima effettuati approfonditi security audit, per verificare che non presenti evidenti vulnerabilità di sicurezza che potrebbero compromettere il sistema per intero.

[1] Annunciato nel README al link `https://github.com/cuckoosandbox/cuckoo/commit/50452a39ff7c3e0c4c94d114bc6317101633b958`

Bibliografia

[1] Bake, K. The 12 most common types of malware (2023). Available from: https://www.crowdstrike.com/cybersecurity-101/malware/types-of-malware/.

[2] Ballenthin, W. and Raabe, M. capa: Automatically identify malware capabilities. *Mandiant Threat Research*, (2020). https://www.mandiant.com/resources/blog/capa-automatically-identify-malware-capabilities.

[3] Gizzi, L. Cyber kill chain, ecco come identificare un attacco informatico e adottare le giuste contromisure (2019). Available from: https://tinyurl.com/3mzfs7ne.

[4] Pedini, A. What is an executable? In *Forensic Friday Gyala S.r.l. 21-04-2023* (2023).

[5] Shirey, R. W. Internet Security Glossary, Version 2. RFC 4949 (2007). Available from: https://www.rfc-editor.org/info/rfc4949, doi:10.17487/RFC4949.

[6] Souppaya, M. and Scarfone, K. Guide to malware incident prevention and handling for desktops and laptops. Tech. rep., U.S. Department of Commerce (2013). Available from: https://doi.org/10.6028/nist.sp.800-83r1, doi:10.6028/nist.sp.800-83r1.

[7] Strom, B. Att&ck 101 (2018). Available from: https://web.archive.org/web/20220301232425/https://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/attck-101.