# Automazione dell’analisi statica e dinamica di malware

Laurea in Informatica

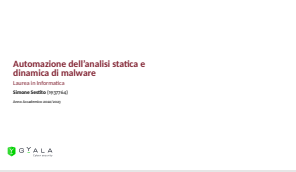
**Simone Sestito** (1937764)

Anno Accademico 2022/2023

GYALA Cyber security

SAPIENZA UNIVERSITÀ DI ROMA






Il progetto si focalizza sull’**automazione** di analisi di malware.
Realizzato nella fattispecie di un tirocinio svolto **in azienda**, e dando come prodotto uno strumento:

- Integrabile con il resto dei sistemi aziendali
- Facilmente scalabile ed estensibile, in ottica futura

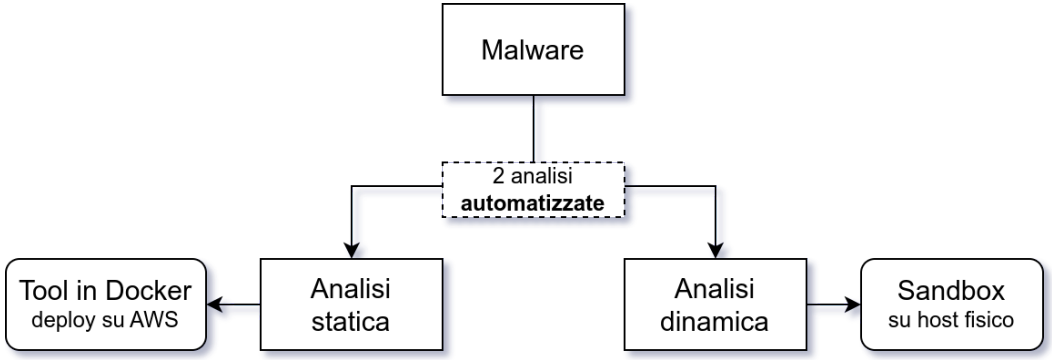



Figure: Struttura del progetto ad alto livello






Figure: Struttura del progetto ad alto livello

Si compone di due aree:

- Statica
  - Dove vengono aggregati e ottimizzati vari tool molto diversi tra loro
  - Eseguito in ambiente sicuro containerizzato
  - Deploy dello strumento in cloud su AWS
- Dinamica
  - Dove viene configurata una sandbox
  - Adeguamente isolata
  - Estesa
  - Resa facilmente portabile su altri sistemi

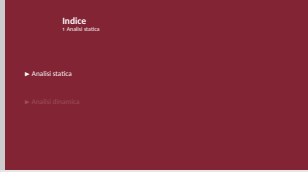
# Indice












Quindi iniziamo con la parte **statica**

# Estrazione delle capabilities

1 Analisi statica

Usiamo *capa*, strumento open source realizzato da Mandiant.





Estrazione delle capabilities
1 Analisi statica
Usiamo *capa*, strumento open source realizzato da Mandiant.

Dato un file eseguibile, prima di tutto, siamo interessati alle sue capabilities. Queste sono le possibili azioni che il programma è in grado di compiere. Ad esempio, se può leggere/scrivere file, modificare registri, eseguire operazioni crittografiche, e così via.
Le andiamo ad estrarre usando capa, ossia un tool open source di Mandiant, nonché uno dei più potenti disponibili

---

Ha inoltre il grande vantaggio di permettere l'integrazione di regole YAML che possono essere sia reperite anche da repository pubblici di altre società o ricercatori, senza necessità di modificare il tool ma solo aggiungere file di regole

---

In più, dobbiamo fare il parsing dell'output che fornisce capa, che ovviamente nasce in forma testuale e tabellare, come nella foto, perché è fatto per essere usato interattivamente. Quindi si va ad ottenere un JSON, quindi un formato machine-readable, eseguendo questo parsing.


# Estrazione delle capabilities

## 1 Analisi statica

Usiamo *capa*, strumento open source realizzato da Mandiant.

- Flessibilità con regole custom in YAML

Dato un file eseguibile, prima di tutto, siamo interessati alle sue capabilities. Queste sono le possibili azioni che il programma è in grado di compiere. Ad esempio, se può leggere/scrivere file, modificare registri, eseguire operazioni crittografiche, e così via.
Le andiamo ad estrarre usando capa, ossia un tool open source di Mandiant, nonché uno dei più potenti disponibili

---

Ha inoltre il grande vantaggio di permettere l'integrazione di regole YAML che possono essere sia reperite anche da repository pubblici di altre società o ricercatori, senza necessità di modificare il tool ma solo aggiungere file di regole

---

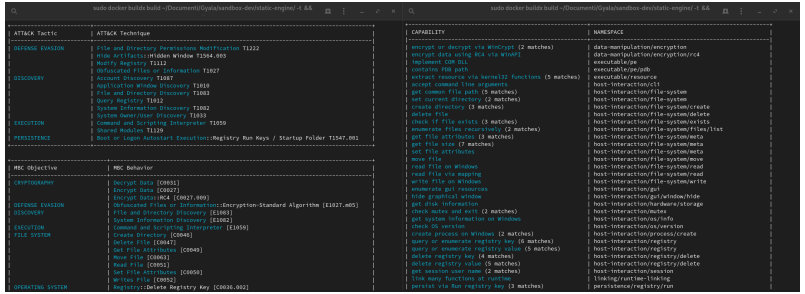
In più, dobbiamo fare il parsing dell'output che fornisce capa, che ovviamente nasce in forma testuale e tabellare, come nella foto, perché è fatto per essere usato interattivamente. Quindi si va ad ottenere un JSON, quindi un formato machine-readable, eseguendo questo parsing.

# Estrazione delle capabilities

1 Analisi statica

Usiamo *capa*, strumento open source realizzato da Mandiant.

- Flessibilità con regole custom in YAML
- Successivo parsing dell'output tabellare da testo a JSON machine-readable








Estrazione delle capabilities
1 Analisi statica
Usiamo capa, strumento open source realizzato da Mandiant.
- Flessibilità con regole custom in YAML
- Successivo parsing dell'output tabellare da testo a JSON machine-readable

Dato un file eseguibile, prima di tutto, siamo interessati alle sue capabilities. Queste sono le possibili azioni che il programma è in grado di compiere. Ad esempio, se può leggere/scrivere file, modificare registri, eseguire operazioni crittografiche, e così via.
Le andiamo ad estrarre usando capa, ossia un tool open source di Mandiant, nonché uno dei più potenti disponibili

Ha inoltre il grande vantaggio di permettere l'integrazione di regole YAML che possono essere sia reperite anche da repository pubblici di altre società o ricercatori, senza necessità di modificare il tool ma solo aggiungere file di regole

In più, dobbiamo fare il parsing dell'output che fornisce capa, che ovviamente nasce in forma testuale e tabellare, come nella foto, perché è fatto per essere usato interattivamente. Quindi si va ad ottenere un JSON, quindi un formato machine-readable, eseguendo questo parsing.

# Casi non supportati

1 Analisi statica

Capabilities estratte tramire analisi **statica** del codice.
Non sempre funziona (packer, installer, runtime come Visual Basic).
capa terminerà con **exit code 14**





Casi non supportati
1 Analisi statica
Capabilities estratte tramire analisi **statica** del codice.
Non sempre funziona (packer, installer, runtime come Visual Basic).
capa terminerà con **exit code 14**

Uno dei più grandi però è la sua fragilità. Infatti l'estrazione delle capabilities avviene tramite un'analisi statica del codice che però non funziona nei casi di eseguibile pacchettizzato, installer o con runtime particolari (es: Visual Basic). Perchè in questi casi, anche se si tratta di una semplice decompressione a runtime del vero codice dell'eseguibile, resta molto difficile avere risultati che siano affidabili.

E in questi casi, capa rileva la situazione, terminando con exit code 14.

- Può portare a un fallimento dell'intero processo di analisi statica
- Ma anche se gestito con try/catch... Crea spreco di risorse (CPU e memoria) per diversi minuti

Quindi si va a realizzare tutta una costruzione attorno per renderlo più adatto alle nostre esigenze

# Casi non supportati

1 Analisi statica

Capabilities estratte tramire analisi **statica** del codice.
Non sempre funziona (packer, installer, runtime come Visual Basic).
capa terminerà con **exit code 14**

- Crash del processo di analisi
- Spreco di risorse





Casi non supportati
1 Analisi statica
Capabilities estratte tramire analisi **statica** del codice.
Non sempre funziona (packer, installer, runtime come Visual Basic).
capa terminerà con **exit code 14**
- Crash del processo di analisi
- Spreco di risorse

Uno dei più grandi però è la sua fragilità. Infatti l'estrazione delle capabilities avviene tramite un'analisi statica del codice che però non funziona nei casi di eseguibile pacchettizzato, installer o con runtime particolari (es: Visual Basic). Perchè in questi casi, anche se si tratta di una semplice decompressione a runtime del vero codice dell'eseguibile, resta molto difficile avere risultati che siano affidabili.

E in questi casi, capa rileva la situazione, terminando con exit code 14.

- Può portare a un fallimento dell'intero processo di analisi statica
- Ma anche se gestito con try/catch... Crea spreco di risorse (CPU e memoria) per diversi minuti

Quindi si va a realizzare tutta una costruzione attorno per renderlo più adatto alle nostre esigenze

# Rilevazione del caso non supportato

1 Analisi statica

Capa ha regole che rilevano questi casi, ma eseguirle richiede **troppi minuti.**

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

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

Dobbiamo creare un **rilevatore custom** da zero, basandoci sulle regole.

Ma come fa a rilevarlo? All'interno sono presenti delle regole anche per questo scopo. Il problema però è che la loro esecuzione di default va a richiedere troppo tempo, e vogliamo ridurlo.

La nostra rilevazione durerà adesso meno di un secondo, a fronte degli svariati minuti che impiegava il tool, risparmiando in termini di risorse ma di conseguenza anche economicamente. Questo perché andiamo a sfruttare che si tratti solo di rilevare la presenza di particolari sezioni o stringhe, senza avviare l'intero processo di feature extraction

# YARA

## 1 Analisi statica

Si esegue anche signature-based matching con lo strumento YARA.

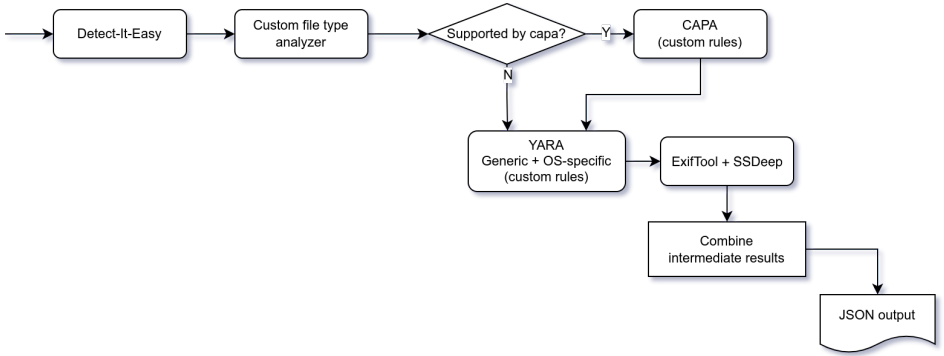
```
simone@archlinux:~/Documenti/Gyala/sandbox-dev/static-engine...
→ yara-rules git:(develop) for d in *; do printf "Rules in $d/: ";
find "$d" -type f | wc -l; done
Rules in common/: 187
Rules in linux/: 206
Rules in osx/: 33
Rules in unclassified/: 620
Rules in windows/: 675
→ yara-rules git:(develop)
```

Figure: Divisione delle regole per OS per ottimizzazione

Successivamente, si è interessati anche ad analisi basate su signature matching, implementate mediante regole YARA.
Queste sono collezionabili da varie realtà della cybersecurity (aziende, ricercatori, ...)
oltre a tutte quelle sviluppate internamente
Per ottimizzare l'esecuzione, le regole sono state divise in cartelle ed eseguite solo quando necessario, in base al sistema operativo target, così da omettere quelle inutili

# Flusso di esecuzione

1 Analisi statica








Nel flusso di esecuzione sono integrate anche altre tecniche minori:

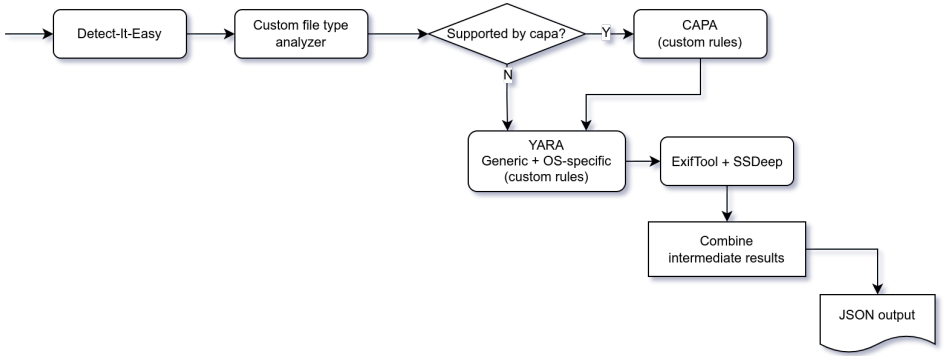
- **SSDeep** per fuzzy hashing e correlazione di sample simili tra loro
- **Detect-It-Easy** e **ExifTool** estraggono ulteriori metadati dal file in analisi (linker utilizzato, descrizioni, charset, tabella dei simboli, ...)

Tutti integrati in un *unico flusso di esecuzione* e uniti in uno stesso risultato JSON finale, continuando ad eseguire parsing e trasformazioni di ciò che otteniamo dagli strumenti grezzi

La gestione degli errori avviene tramite handle degli exit-code e restituzione di un JSON contenente il campo *"error"* con un error code tra le stringhe predeterminate

# Flusso di esecuzione

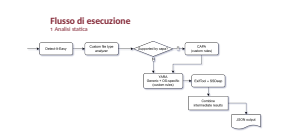
1 Analisi statica




Errori gestiti tramite handle degli exit-code, garantendo sempre un JSON valido in output








Nel flusso di esecuzione sono integrate anche altre tecniche minori:

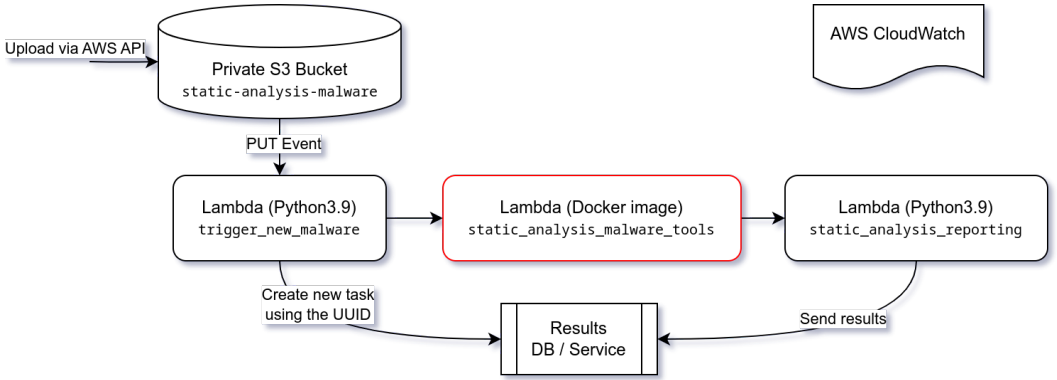
- **SSDeep** per fuzzy hashing e correlazione di sample simili tra loro
- **Detect-It-Easy** e **ExifTool** estraggono ulteriori metadati dal file in analisi (linker utilizzato, descrizioni, charset, tabella dei simboli, ...)

Tutti integrati in un *unico flusso di esecuzione* e uniti in uno stesso risultato JSON finale, continuando ad eseguire parsing e trasformazioni di ciò che otteniamo dagli strumenti grezzi

La gestione degli errori avviene tramite handle degli exit-code e restituzione di un JSON contenente il campo *"error"* con un error code tra le stringhe predeterminate

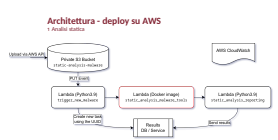
# Architettura - deploy su AWS

1 Analisi statica







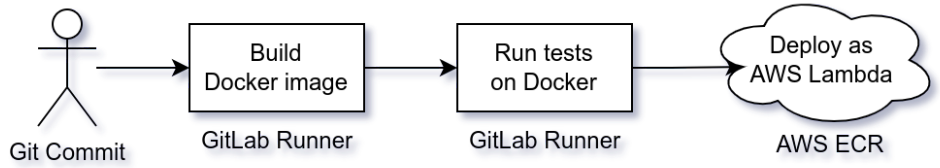
Una volta fatto il deploy in cloud su AWS, questo è il flusso di esecuzione a più alto livello, dove ciò che abbiamo appena visto è tutto all'interno della Lambda malware_tools, distribuita come un'immagine Docker.

Dopo un'analisi dei costi e delle caratteristiche di AWS Lambda, EC2 e simili, si è optato per eseguire questa analisi su AWS Lambda. E qui ne vediamo 3 distinte, non una sola. Questo perché si creano varie Lambda per minimizzare i permessi assegnati dove viene effettivamente analizzato il malware, seguendo il principio del *Least Privilege*

# Pipeline CI/CD

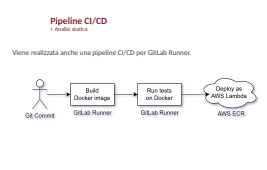
1 Analisi statica

Viene realizzata anche una pipeline CI/CD per GitLab Runner.








Per automatizzare anche il processo di deploy dello strumento, si è creata una pipeline CI/CD su GitLab che esegue in appositi Runner.

Così, se si modifica una regola o esegue una modifica, al push del commit Git viene eseguito: **GLI STEP**

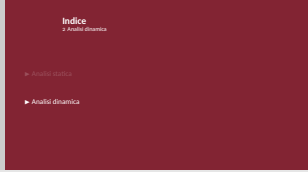
# Indice







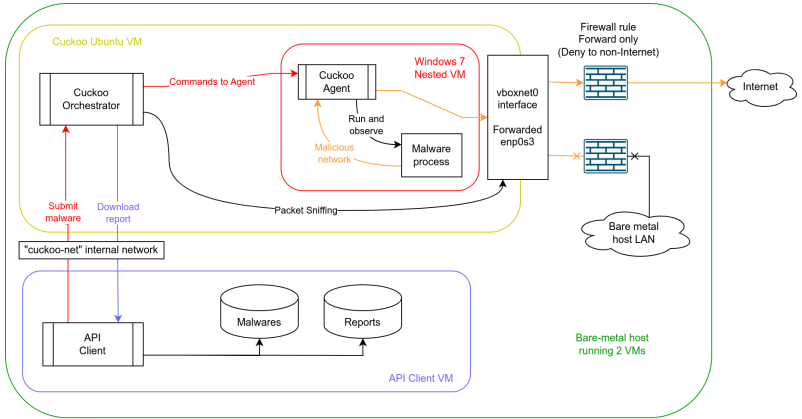
Spostiamoci ora sulla parte dinamica



# Cosa è stato realizzato

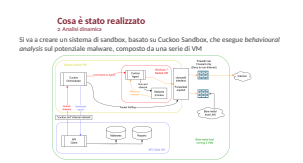
## 2 Analisi dinamica

Si va a creare un sistema di sandbox, basato su Cuckoo Sandbox, che esegue *behavioural analysis* sul potenziale malware, composto da una serie di VM







Si compone di:

- una serie di macchine virtuali Virtualbox, anche innestate,
- poi configurate per massimizzare la sicurezza e l'isolamento,
- infine estese con altri plugin Python

La lunga configurazione del sistema ha previsto anche realizzazione di systemd unit, uso di Docker Compose per le dipendenze su servizi terzi (es: database) e uso di Virtualenv per poter usare Python 2 su VM Ubuntu.

# Anti VM-detection (1)

2 Analisi dinamica

Si rende il sistema **verosimile**, installando programmi di utilità o molto diffusi, runtime, risorse come un computer scarso ma reale.

**E tecniche più sofisticate**
Quali mitigazioni con DLL injection, uso di agent con OCR per simulare l'interazione umana, …

Superiamo *quasi* ogni test di pafish (strumento per testare la VM detectability)





Anti VM-detection (1)
2 Analisi dinamica
Si rende il sistema **verosimile**, installando programmi di utilità o molto diffusi, runtime, risorse come un computer scarso ma reale.
**E tecniche più sofisticate**
Quali mitigazioni con DLL injection, uso di agent con OCR per simulare l'interazione umana, …
Superiamo *quasi* ogni test di pafish (strumento per testare la VM detectability)

Per rendere l'ambiente più verosimile possibile:

- installiamo tutti programmi normalmente presenti, come browser, suite Office, PDF reader, WinRAR, …
- allocando risorse di sistema reali, quali 4GB di RAM e 128GB di disco rigido
- installazione di runtime: service pack 1, Visual C++ redistributable, …

Un malware potrebbe vedere l'assenza di un software sempre presente (es: Chrome) e decidere di non eseguire il reale comportamento malevolo, sospettando di essere analizzato

## Plugin

2 Analisi dinamica

I tool integrati sono:

- **Patriot**, che rileva tecniche *in-memory stealth*
- **Hunt-Sleeping-Beacons** per tattiche particolari di sleep obfuscation





Plugin
2 Analisi dinamica
I tool integrati sono:
• Patriot, che rileva tecniche in-memory stealth
• Hunt-Sleeping-Beacons per tattiche particolari di sleep obfuscation

è stata estesa la potenza della sandbox sviluppando dei plugin in Python per aggiungere altri tool, di per sè non supportati.

- **Patriot**, rileva tecniche di occultamento in memoria (*in-memory stealth*), usate anche da Cobalt Strike
- **Hunt-Sleeping-Beacons**, identifica processi che stanno in stato di Sleep ma in modi non standard e non rilevabili tradizionalmente, come *Foliage*; quindi contro sleep obfuscation

.

.

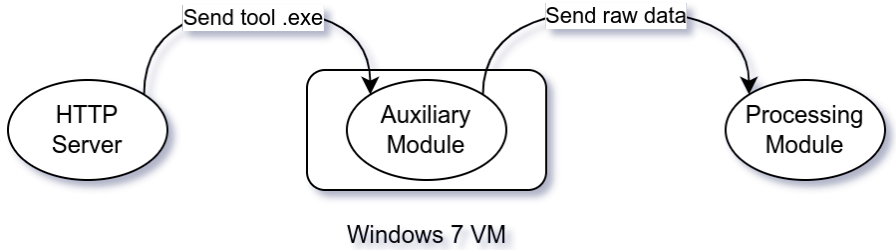
Questo apre anche le porte a una futura integrazioni di strumenti sempre più recenti, basterà creare un plugin seguendo questo stesso schema. **COMPONENTI SLIDE**

## Plugin
2 Analisi dinamica

I tool integrati sono:

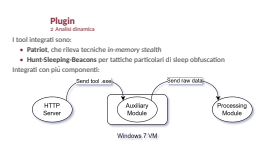
- **Patriot**, che rileva tecniche *in-memory stealth*
- **Hunt-Sleeping-Beacons** per tattiche particolari di sleep obfuscation

Integrati con più componenti:








è stata estesa la potenza della sandbox sviluppando dei plugin in Python per aggiungere altri tool, di per sè non supportati.

- **Patriot**, rileva tecniche di occultamento in memoria (*in-memory stealth*), usate anche da Cobalt Strike
- **Hunt-Sleeping-Beacons**, identifica processi che stanno in stato di Sleep ma in modi non standard e non rilevabili tradizionalmente, come *Foliage*; quindi contro sleep obfuscation

.

.

Questo apre anche le porte a una futura integrazioni di strumenti sempre più recenti, basterà creare un plugin seguendo questo stesso schema. **COMPONENTI SLIDE**

# Hardening
## 2 Analisi dinamica

Si adottano tecniche per fare l'hardening della sandbox:

- Firewall, per proteggere a livello di **rete**
  - a livello di **rete** viene usato un firewall per impedire che la sandbox possa contattare altri dispositivi nella rete dell'host, servizi dell'agent che non dovrebbero essere esposti, e forwarding dei pacchetti dalla sandbox all'interfaccia di rete sotto sniffing indirizzati verso Internet

---

- a livello di **utente** viene creato uno Unix user senza login shell e con i minimi privilegi per ridurre la superficie di attacco se compromessa la VM dove esegue Cuckoo

---

- ripristino **snapshot** della VM Windows dopo ogni analisi per ripristinare sempre lo stato pulito e iniziale

---

- creazione di un'**altra VM** su rete interna (non connessa a Internet) per interagire con la VM sandbox e salvare sample e risultati

# Hardening
## 2 Analisi dinamica

Si adottano tecniche per fare l'hardening della sandbox:

- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo





Hardening
2 Analisi dinamica
Si adottano tecniche per fare l'hardening della sandbox:
- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo

- a livello di **rete** viene usato un firewall per impedire che la sandbox possa contattare altri dispositivi nella rete dell'host, servizi dell'agent che non dovrebbero essere esposti, e forwarding dei pacchetti dalla sandbox all'interfaccia di rete sotto sniffing indirizzati verso Internet

---

- a livello di **utente** viene creato uno Unix user senza login shell e con i minimi privilegi per ridurre la superficie di attacco se compromessa la VM dove esegue Cuckoo

---

- ripristino **snapshot** della VM Windows dopo ogni analisi per ripristinare sempre lo stato pulito e iniziale

---

- creazione di un'**altra VM** su rete interna (non connessa a Internet) per interagire con la VM sandbox e salvare sample e risultati

# Hardening

## 2 Analisi dinamica

Si adottano tecniche per fare l'hardening della sandbox:

- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo
- ripristino **snapshot** dopo ogni analisi





Hardening
2 Analisi dinamica
Si adottano tecniche per fare l'hardening della sandbox:
- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo
- ripristino **snapshot** dopo ogni analisi

- a livello di **rete** viene usato un firewall per impedire che la sandbox possa contattare altri dispositivi nella rete dell'host, servizi dell'agent che non dovrebbero essere esposti, e forwarding dei pacchetti dalla sandbox all'interfaccia di rete sotto sniffing indirizzati verso Internet

---

- a livello di **utente** viene creato uno Unix user senza login shell e con i minimi privilegi per ridurre la superficie di attacco se compromessa la VM dove esegue Cuckoo

---

- ripristino **snapshot** della VM Windows dopo ogni analisi per ripristinare sempre lo stato pulito e iniziale

---

- creazione di un'**altra VM** su rete interna (non connessa a Internet) per interagire con la VM sandbox e salvare sample e risultati

# Hardening
## 2 Analisi dinamica

Si adottano tecniche per fare l'hardening della sandbox:

- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo
- ripristino **snapshot** dopo ogni analisi
- creazione di un'**altra VM** su rete interna





Hardening
2 Analisi dinamica
Si adottano tecniche per fare l'hardening della sandbox:
- Firewall, per proteggere a livello di **rete**
- Utente Linux separato per eseguire Cuckoo
- ripristino **snapshot** dopo ogni analisi
- creazione di un'**altra VM** su rete interna

- a livello di **rete** viene usato un firewall per impedire che la sandbox possa contattare altri dispositivi nella rete dell'host, servizi dell'agent che non dovrebbero essere esposti, e forwarding dei pacchetti dalla sandbox all'interfaccia di rete sotto sniffing indirizzati verso Internet

---

- a livello di **utente** viene creato uno Unix user senza login shell e con i minimi privilegi per ridurre la superficie di attacco se compromessa la VM dove esegue Cuckoo

---

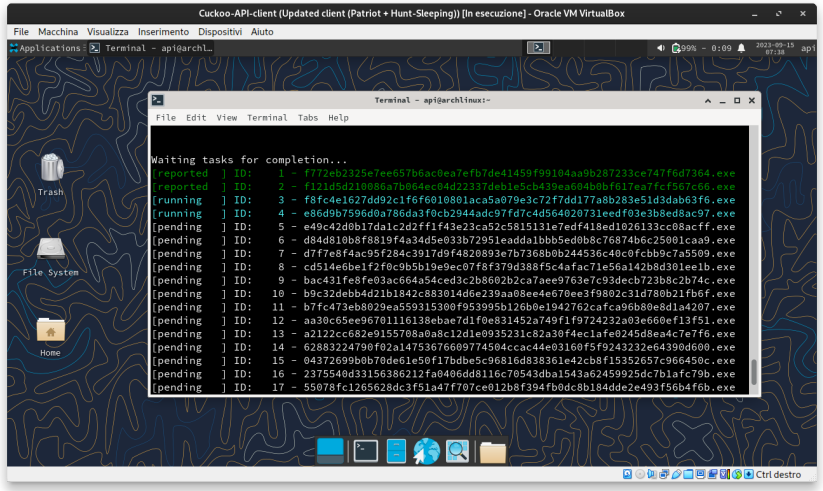
- ripristino **snapshot** della VM Windows dopo ogni analisi per ripristinare sempre lo stato pulito e iniziale

---

- creazione di un'**altra VM** su rete interna (non connessa a Internet) per interagire con la VM sandbox e salvare sample e risultati

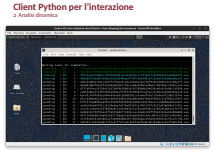
# Client Python per l'interazione

2 Analisi dinamica







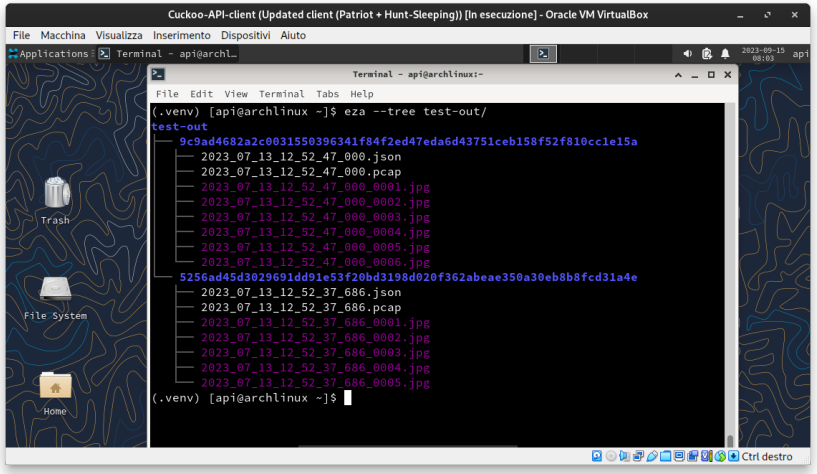



Per interagire con la sandbox, si è creato un client Python. Qui vediamo lo screenshot della sua esecuzione in modalità interattiva.

1. Astrae l'uso della REST API di Cuckoo tramite delle apposite classi (può essere riciclato il codice in futuro, riducendo il rischio di introdurre bug derivanti da una nuova implementazione)
2. Fornisce un'interfaccia utente all'analista più semplice di una web UI (che è stata limitata in fase di Hardening)

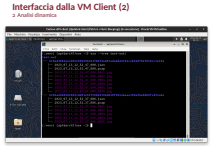
# Interfaccia dalla VM Client (2)

2 Analisi dinamica







Il report finale viene trasformato da ciò che fornirà Cuckoo di default. Vengono:

- divise le syscall per thread, e non per processo
- rimosse informazioni duplicate o poco chiare, organizzandole al meglio
- ristrutturate le informazioni sul traffico di rete, dividendole per protocollo o categoria

Inoltre sono presenti screenshot, file creati dal malware e file PCAP della cattura dei pacchetti di rete

# Automazione dell’analisi statica e dinamica di malware

*Grazie per l’attenzione!*
*Domande?*





Automazione dell'analisi statica e dinamica di malware

*Grazie per l'attenzione!*
*Domande?*