Università degli Studi di Messina
MIFT
Corso di Laurea in Informatica

Relazione progetto Laboratorio di
Amministrazione dei sistemi

Servizio che rileva attività sospetta e notifica tramite mail

Realizzato da
Giovanni Amendolia
Thilina Palihawadana Arachchige Perera

Anno Accademico 2022/2023

Contents

1 Introduzione

1.1 Idea di progetto

Il progetto prevede la creazione di un demone systemd che monitora i tentativi di accesso al sistema e l'esecuzione di operazioni privilegiate, e notifichi gli amministratori tramite mail[1] se vengono rilevati tentativi d'accesso o di elevazione di privilegi sospetti. Questi controlli vengono fatti leggendo e parsando il file di **auth.log**.

Vengono considerati come attività sospetta:

- tentativi di login falliti consecutivi a un qualsiasi account
- tentativi di esecuzione di comandi privilegiati tramite **sudo**
- tentativi di cambio utente tramite **su**, verso qualsiasi account
- tentativi di accesso effettuati in certe fasce orarie (ad esempio la notte)

Il servizio è configurabile tramite il comando **nanoalerdmail**, che apre il file di configurazione tramite nano, solo da utenti con privilegi. La configurazione prevede:

- indirizzi email a cui mandare i messaggi di alert
- parametri per identificare l'attività sospetta, quindi numero consecutivo di tentativi falliti e intervallo di tempo in cui sono avvenuti

Il servizio inoltre scrive sul file di log **/var/log/alertmaild.log** le attività svolte tra cui: avvio del demone, arresto del demone e invio di mail.

Tramite un Makefile viene creato un package **.deb** per cui installare il servizio includendo gli script.

Tutti i file di installazione e configurazione e gli script sono disponibili sulla repo GitHub[2]

2 Implementazione

Il servizio è stato implementato in Python, in modo da essere più portatile e flessibile, ma soprattutto per l'ampio insieme di moduli disponibili e attività della community.

Nelle prossime sezioni verranno analizzate singolarmente le componenti principali del progetto, partendo dalla creazione del package .deb fino alle specifiche del codice.

[2] Per evitare complessità dovute all'ottenimento di un dominio e il riconoscimento di questo presso server mail già esistenti come ad esempio Google, le mail possono essere inviate solo a dei domini di mail temporanee.

[2] https://github.com/Cestnut/alertmaild/tree/main

2.1 Makefile

Per creare il package alertmaild.deb è stato utilizzato un Makefile. Il Makefile è un file letto da **make**. un'utility per facilitare la creazione di file e di esecuzione di comandi su di essi tramite la creazione di target.

Di seguito il Makefile utilizzato:

```
CC=gcc
CFLAGS= -Wall

all: deb clean

compile:nanoalertmaild.c
	$(CC) $(CFLAGS) nanoalertmaild.c -o nanoalertmaild


deb: compile
	mkdir tmp
	mkdir -p tmp/opt/
	mkdir -p tmp/lib/systemd/system
	mkdir -p tmp/var/log/
	mkdir -p tmp/usr/bin

	cp -r alertmaild tmp/opt/
	cp alertmail.service tmp/lib/systemd/system
	touch tmp/var/log/alertmaild.log
	cp nanoalertmaild tmp/usr/bin

	cp -r DEBIAN/ tmp/

	dpkg-deb --build tmp alertmaild.deb

clean:
	rm -r tmp
	rm nanoalertmaild
```

Vengono creati quattro target: **all**, **compile**,**deb** e **clean**. Invocando **make** o **make all** vengono invocati in ordine il target deb, che compila l'eseguibile nanoalertmaild e costruisce la directory temporanea **tmp** da cui crea il package .deb, e il target clean che rimuove la cartella **tmp** e l'eseguibile nanoalertmaild dato che non servono più.

Il package .deb viene creato usando il comando

```
	dpkg-deb --build tmp alertmaild.deb
```

dove tmp è la directory da cui partire, in cui vengono create tutte le directory

e copiati i file. Durante l'installazione del package, tutti i file e le directory vengono creati partendo dalla root.

Una directory particolare è la directory DEBIAN in cui vengono inseriti i file che verranno usati dall'installer. Per questo progetto sono stati creati i file **control** contenente metadati sul package, **postinst** script che viene avviato al termine dell'installazione e **prerm** che verrà avviato prima della rimozione del package.

Particolarmente interessante è lo script bash postinst, di seguito il contenuto.

```
#!/bin/bash

cd /opt/alertmaild
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

chown -R root:root /opt/alertmaild
chmod -R 700 /opt/alertmaild

chown root:root /usr/bin/nanoalertmaild
chmod 700 /usr/bin/nanoalertmaild

systemctl enable alertmail
systemctl start alertmail
```

Lo script crea un ambiente virtuale python nella directory contente il codice del demone in cui installa le dipendenze, elencate nel file requirements.txt cioè **secure-smtplib** e **ConfigParser**. Dopodiché cambia owner e permessi della cartella del servizio e dell'eseguibile nanoalertmaild, in modo che siano eseguibili solo da chi dispone di permessi di root.

Infine attiva ed avvia il servizio systemd.

2.2 Container SMTP

Il server SMTP viene deployate tramite container utilizzando docker compose. Di seguito il file **docker-compose.yml**

```
version: '3'
services:
  mail:
    container_name: "mail"
    image: bytemark/smtp
    ports:
      - "25:25"
```

Vengono specificate:

- La versione di docker compose
- Il nome del container
- L'immagine da utilizzare
- Le porte su cui fare il binding

2.3 Service Unit systemd

Systemd è un sistema di inizializzazione e gestione dei processi ampiamente utilizzato nei sistemi operativi basati su Linux. È stato progettato per sostituire il tradizionale sistema di avvio init e offre una serie di funzionalità avanzate per la gestione dei servizi, dei processi e delle risorse di sistema.

Un concetto chiave in systemd sono gli "unit file". Gli unit file sono dei file di configurazione che descrivono come un servizio, un dispositivo, un socket o altre risorse dovrebbero essere gestite da systemd. Ogni unit file definisce un "unit", che è un'istanza di qualcosa che systemd può gestire. Gli unit file specificano varie informazioni, come il nome dell'unità, il percorso dell'eseguibile, gli argomenti da passare al processo, le dipendenze da altre unità e molte altre opzioni di configurazione.

In particolar modo per la creazione di demoni vengono utilizzati i Service Units che descrivono i servizi o i demoni che vengono eseguiti in background. Il file **alertmaild.service** è così composto:

```
[Unit]
Description=Alerter mail service
After=network.target
After=docker.service
StartLimitIntervalSec=0

[Service]
```

```
WorkingDirectory=/opt/alertmaild
ExecStart=/opt/alertmaild/venv/bin/python3 /opt/alertmaild/mail_sender.py
Restart=always


[Install]
WantedBy=default.target
```

Come si può vedere è composto da tre sezioni principali:

- **Unit** definisce i metedata dell'unit e le relazioni tra l'unit e le altre unit.
  - After: Le unità vengono avviate prima di quella attuale
  - StartLimitIntervalSec: quante volte provare a riavviare prima di smettere di provare. Il valore 0 indica che possono essere fatti illimitati tentativi.
- **Service** configurazione solo dei servizi (le altre due possono essere applicate a qualsiasi tipo di unit)
  - ExecStart, il comando da eseguire all'avvio del servizio
  - Restart, in quale condizione riavviare il servizio al termine dell'esecuzione (nel nostro caso sempre)
- **Install**
- Sezione presente e utilizzata solo dai servizi in stato di *enabled*
  - WantedBy, lega l'unit con un target. Quando quel target viene avviato, anche tutte le unit legate a quel target vengono avviate.

2.4 File di configurazione e nanoalertmaild

Le impostazioni del servizio si trovano nel file di configurazione, che può essere aperto manualmente o tramite l'eseguibile nanoalertmaild.

Il file di configurazione ha la seguente struttura:

```
[GENERALE]
SMTPhost=localhost
SMTPport=25
MailingList=dipejih139@pbridal.com:nofitar605@trazeco.com
MailSource=alertmaild@companydomain.com


FasciaOrariaBool=False
FasciaOrariaIntervalloMinuti=60
FasciaOrariaInizio=00:00
```

```
FasciaOrariaFine=23:00

[sudo]
TentativiFallitiConsecutiviNumero=3

[su]
TentativiFallitiConsecutiviNumero=1

[login]
TentativiFallitiConsecutiviNumero=3
```

Spieghiamo adesso i campi di maggiore rilevanza:

- **MailingList**, la lista di mail a cui mandare gli alert, separate dai due punti
- **MailSource**, la mail da usare come mittente
- **FasciaOrariaBool**, se effettuare il controllo sulla fascia oraria
- **FasciaOrariaIntervalloMinuti**, numero di minuti che devono passare prima dell'ultimo alert per evento avvenuto in una determinata fascia oraria prima di inviarne un altro
- **FasciaOrariaBoolInizio** e **FasciaOrariaFine**, estremi del range orario in cui l'attività viene considerata sospetta, formato %H:%M
- **TentativiFallitiConsecutiviNumero**, per ogni comando monitorato, dopo quanti tentativi falliti inviare l'alert (a ogni tentativo riuscito il counter viene azzerato)

Il codice sorgente dell'eseguibile nanoalertmaild è stato scritto in C.

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    const char *filePath = "/opt/alertmaild/alertmaild.conf";

    char command[100];
    snprintf(command, sizeof(command), "nano %s", filePath);

    system(command);

    return 0;
}
```

2.5 Codice Python

Come già detto il codice è stato scritto in Python ed è composto da uno script principale: **mail_sender.py** e la classe **AuthChecker.py** che effettua il controllo attivo sul file **auth.log**.

2.5.1 Lettura file di configurazione

Il file di configurazione viene letto usando il modulo ConfigParser. Il codice rilevante è il seguente:

```
    from configparser import ConfigParser
    ...
    class AlerterMail:
        def __init__(self):
            self.config = ConfigParser()
            self.config.read("alertmaild.conf")

            self.mailing_list = self.config["GENERALE"]["MailingList"].split(":")
            ...
```

L'ultima linea di codice è un esempio di come ottenere i valori dal file di configurazione. In questo caso viene creata una lista partendo dal valore letto dal campo MailingList nella sezione GENERALE.

2.5.2 Invio mail

Per inviare le mail tramite SMTP viene usato il modulo smtplib. Di seguito il codice rilevante su come vengono inviate le mail:

```
    import smtplib
    ...

        message = """From: Computer <{}>
Subject: Alert

{}
"""
        smtpObj = smtplib.SMTP(host=host, port=port)

        #Per ogni alert in alerts(), il generatore di AuthChecker,
        #se l'alert contiene un messaggio invia la mail a tutti i destinatari
        for alert in self.auth_checker.alerts():
            if alert != '':
                receivers = ",".join(self.mailing_list)
```

```
                msg = message.format(sender, alert)
                smtpObj.sendmail(sender, receivers, msg)

                self.log("Sent mail. Content: {}".format(alert))
```

2.5.3 reports.json

Per tenere traccia dei tentativi falliti viene usato un dizionario, che ha la seguente struttura:

```
{
'login':
    {'giov':
        {'begin_date': 'Aug 26 17:320:23',
        'number': 3}},
 'su':
    {'giov':
        {'root':
            {'number': 5,
            'begin_date': 'Aug 26 17:30:23'}}},
 'sudo':
    {'giov':
        {'begin_date': 'Aug 26 17:38:23',
        'number': 3}},
 'time_range': 0}
```

Di ogni comando tiene traccia chi sia stato ad eseguirlo, quanti tentativi falliti sono stati effettuati e a quando risale il primo tra questi. Riguardo il comando su, come si può vedere, per ogni utente che esegue il comando si tiene un conteggio diverso per ogni utente destinatario.

Dato che il servizio potrebbe essere riavviato, causando la perdita dei dati, nel metodo **stop** della classe **AlerterMail** è stata inserita questa linea di codice:

```
def stop(self):
        with open("reports.json", "w") as reports_file:
            json.dump(self.auth_checker.reports, reports_file)
```

Salva il dizionario in formato json sul file **reports.json**, che viene letto a ogni avvio del servizio, se presente.

2.5.4 Parsing auth.log

Il parsing di auth.log avviene nella classe AuthChecker. Dato che i metodi di controllo sono tanti per ogni comando, ma svolgono più o meno lo stesso lavoro, verranno illustrate solo le funzioni riguardo **su**.

Innanzitutto esponiamo il metodo **follow**, un generatore che ritorna ogni linea nuova del file auth.log.

```
def follow(self, file):
        file.seek(0, os.SEEK_END)
        while True:
            line = file.readline()
            if not line:
                time.sleep(1) #sleep per dare il tempo al file di aggiornarsi
                continue

            yield line
```

Il metodo principale itera sul generatore **follow** e per ogni nuova linea effettua i controlli:

```
def alerts(self):
        auth_file = open("/var/log/auth.log", "r")
        for line in self.follow(auth_file):
            line = line.split(" ")
            line = [i.strip() for i in line if i != ""]
            date = " ".join(line[0:3])
            content = line[4:]
            message = ""
            ...
            elif "su" in content[0]:
                if len(content) > 1 and  "FAILED" in content[1]:
                    message = self.su_failed(content, date)
                elif len(content) > 3 and  "opened" in content[3]:
                    message = self.su_clean(content)
            ...
            yield message
```

Come si può notare vengono invocati due metodi a seconda dei casi; il metodo **su_failed** se viene rilevato un tentativo d'accesso fallito, e il metodo **su_clean** in caso di tentativo d'accesso riuscito, in modo da eventualmente azzerare i contatori.

Metodo **su_failed**:

```
def su_failed(self, content, date):
        src_user = content[5]
        dst_user = content[4].strip(")")

        if src_user not in self.reports["su"]:
            self.reports["su"][src_user] = dict()
```

```
        if dst_user in self.reports["su"][src_user]:
            self.reports["su"][src_user][dst_user]["number"] += 1
        else:
            self.reports["su"][src_user][dst_user] = dict()
            self.reports["su"][src_user][dst_user]["number"] = 1
            self.reports["su"][src_user][dst_user]["begin_date"] = date

        attempt_number = self.reports["su"][src_user][dst_user]["number"]
        max_number = int(self.config["su"]["TentativiFallitiConsecutiviNumero"])
        if attempt_number >= max_number:
            begin_date = self.reports["su"][src_user][dst_user]["begin_date"]
            self.reports["su"][src_user].pop(dst_user)
            return "Consecutive failed su attempt for ..."
        else:
            return ""
```

Semplicemente si occupa di aggiungere e inizializzare i campi al dizionario **self.reports** se non esistono, altrimenti incrementa il contatore. E infine effettua il controllo sul numero di tentativi falliti massimo oltre cui va inviata una notifica di alert.

Metodo **su_clean**:

```
def su_clean(self, content):
        src_user_uid = int(content[8].split("uid=")[1].strip(")"))
        dst_user = content[6].split("(uid=")[0]
        src_user = pwd.getpwuid(src_user_uid).pw_name
        if src_user in self.reports["su"] and dst_user in self.reports["su"][src_user]:
            self.reports["su"][src_user].pop(dst_user)
        return ""
```

Analizza il contenuto della linea di log ottenendo l'UID dell'utente che ha effettuato il comando con successo. Dopodiché ottiene l'username dell'utente tramite il modulo **pwd** e rimuove quella chiave dal dizionario **reports["su"]**