Python e pwntools

Una (molto) breve introduzione

Andrea Oliveri
a.k.a IridiumXOR

Modalita’di funzionamento di Python

Esistono 3 modalita’ d’uso del Python (un po’ come per Bash):

- Modalita’ riga di comando: si richiama l’interprete python dalla shell e si esegue direttamente il codice.
  [andrea@wintermute ~]$ python -c ‘s=”Hello World!”; print(s)’

- Modalita’ script: si richiama uno script gia’ composto in precedenza. (*)
  [andrea@wintermute ~]$ python mioscript.py

- Modalita’ interattiva: permette di digitare direttamente comandi ed eseguirli in una shell interattiva.
  [andrea@wintermute ~]$ python
  Type "help", "copyright", "credits" or "license" for more information.
  >>>

(*) Esattamente come accade per gli script Bash e’ possibile inserire in testa allo script lo “shebang” **#!/usr/bin/env python** e, rendendolo eseguibile, sara’ possibile richiamarlo direttamente da shell come fosse un comando standard.

Tipi di variabili

Il Python, e' un linguaggio interpretato e non tipizzato, che dispone di molti tipi di variabili tra cui:

- **interi e float:** **3 4.5 10000 0x1E**
- **stringhe:** **"ciao mondo", '\x41\x42'**
- **liste:** **["prova", 1, ['ciao']]**
- **dizionari:** **(non li vedremo qua)**

Tutte le variabili in Python sono OGGETTI! Dispongono quindi di alcuni metodi per la loro modifica:

```
[andrea@wintermute ~]$ python -c 'print("CIAO MONDO".lower())'
    ciao mondo
```

COMANDO FONDAMENTALE! **help(oggetto)** -> restituisce il manuale dell'oggetto.

Uso delle variabili (1/2)

Le variabili in python NON hanno un tipo definito ed e' permesso l'uso degli operatori matematici anche su non interi. NON ESISTONO I PUNTATORI! (*)

```
A = 1              # QUESTO E' UN COMMENTO!
B = 2              # Assegno l'intero 2 a B
A += B             # A = A + B
B = A * 2
A = "ciao"         # Adesso la variabile A e' una stringa!
B = " pippo"
C = A + B          # Concateno A e B, quindi C = "ciao pippo"


A = 'CIAO' * 2     # A conterra' la stringa "CIAOCIAO"
```

(*) In realta' e' esattamente l'opposto, ogni cosa e' un puntatore...

Uso delle variabili (2/2)

Le stringhe e le liste sono trattabili come “array” alla C: e’ possibile accedere ai singoli elementi e modificarli, aggiungerli, eliminarli…
GLI INDICI COMINCIANO DA 0! (come in C).

```
l = [‘a’, 1, ‘ciao’] # Lista con elementi di vario tipo
print l[0]           # Stampa ‘a’
l.append(‘pippo’)    # Aggiunge in coda a l la stringa “pippo”

s = ‘andrea’         # Una stringa si puo definire con “” oppure ‘’
s[2]                 # e’ il carattere ‘d’ della stringa
```

posso convertire un oggetto in un altro:

```
l2 = list(s)         # l2 sara’ allora una lista [‘a’, ‘n’, ‘d’, ‘r’, ‘e’, ‘a’]
```

Costrutti di controllo

Come ogni linguaggio il Python dispone dei costrutti IF, FOR, WHILE
(non esistono i costrutti DO...WHILE e SWITCH).

ATTENZIONE!
In python le indentazioni (tab o 4 spazi) sono fondamentali, separano i blocchi come le {} in C!

Operatori logici: **== != and or is in not**

```
if CONDIZIONE:
    ----
    ----
elif CONDIZIONE2:
    ----
    ----
else:
    ----
```

```
while CONDIZIONE:
    ----
    ----

while i<3:
    print(i)
    i += 1
```

```
for i in QUALCOSA:
    ----
    ----

lista = [‘uno’, ‘due’]
for elem in lista:
    print(elem)
```

Definire e chiamare una funzione

Sono molto piu' flessibili delle funzioni C in quanto NON si ha obbligo sul tipo dei parametri, ne sul tipo di ritorno e possono essere definite ovunque e in qualsiasi punto del codice!

```
def mia_funzione(a, b):       # ATTENZIONE all'indentazione!
    c = a + b
    return c

x = 1
y = 2
print(mia_funzione(x,y))      # Stampera' 3. Ma posso fare anche…

w = "ciao"
z = " mondo"
print(mia_funzione(w,z))      # Stampera' "ciao mondo" :D
```

print, import e PIP

PRINT:

Per stampare si usa una sintassi simile alla printf() del C

```
a=10
print “a vale: %d” % a
```

oppure

```
a=10
print “a vale: {}”.format(str(a))
```

La seconda forma stampa una stringa formattata sostituendo a {} la “stringhificazione” della variabile a.

IMPORT:

La direttiva import e’ simile alla direttiva #include del C e serve ad importare un modulo di estensione del Python al fine di usarlo nel codice.

```
import json
json.dumps(....)

from json import dumps, loads
dumps(....)
loads(....)

from json import *
dumps(....)
loads(....)
load(....)
```

PIP:

E’ il package manager principale di Python, permette di installare nuovi moduli non presenti di default (e’ sia un comando di sistema sia, a sua volta, un modulo :)

```
# pip search pwntools
# pip install pwntools
```

ATTENZIONE!
NON aggiornate pip da pip stesso (spesso si incorre in errori “fatali”!)

virtualenv

virtualenv e' un comando di sistema (installabile con pip) che permette la creazione di un ambiente virtuale Python "separato" in cui operare. Una volta avviato crea una nuova installazione Python (e relative utility, come pip) in una cartella separata. Cio' permette di sperimentare liberamente (installare, e rimuovere pacchetti ad esempio) senza modificare l'installazione Python di sistema.

$ virtualenv VENV -> Crea un nuovo ambiente virtuale chiamato VENV posizionato all'interno dell'omonima cartella.

$ source VENV/bin/activate -> Attiva l'ambiente virtuale. Da adesso si utilizzera' l'interprete python e i relativi tool dell'ambiente virtuale e non piu' quelli di sistema.

$ deactivate -> Disattiva il virtualenv (non lo rimuove!).

diff python2 python3

Il codice scritto in Py3 e' generalmente compatibile con Py2 a patto che si tenga conto di alcune sottigliezze [fatto salvo l'esistenza delle librerie usate in Py2 che devono esistere anche per Py3 (pwntools NON ESISTE per Py3 :( )]

PRINT:

Py2: **print "ciao"**

Py3: **print("ciao")**

In Py3 print e' una funzione a tutti gli effetti e richiede quindi le parentesi.

STRINGHE:

Py2: le stringhe possono contenere **solo** caratteri **ASCII.**

Py3: le stringhe gestiscono in modo **nativo UTF-8.**

DIVISIONE TRA INTERI:

Py2: la divisione tra interi produce la parte intera del risultato.

3 / 2 = 1
3 / 2.0 = 0.666

Py3: la divisione e' la divisione float

3 / 2 = 0.666
3//2 = 1 (divisione intera)

pwntools

E’ un modulo per **PYTHON2** (non Py3!) che permette di automatizzare molte operazioni utili per exploitare eseguibili e non solo…

Per installare pwntools PER L’INSTALLAZIONE PYTHON DI SISTEMA si utilizza PIP:

```
[andrea@wintermute ~]$ sudo pip2 install pwntools
```

ATTENZIONE!

- Consiglio vivamente di installare pwntools in un ambiente virtuale come spiegato in slide[8].
- Utilizzate il comando pip2 (non il comando pip che potrebbe essere un link a pip3).

pwntools: alcune funzioni utili (1/4)

In pwntools e' possibile definire un **context** di esecuzione, cioe' la "cornice" all'interno della quale lavoreremo. Potremo settare ad esempio, l'architettura del processore e il sistema operativo su cui sara' in esecuzione il nostro target. Pwntools modifichera' automaticamente il proprio comportamento di conseguenza agevolando il nostro lavoro.

```
from pwn import *

context(arch='i386', os='linux')
```

pwntools: alcune funzioni utili (2/4)

Come interfacciarsi ad un processo per eseguirne l'exploitation?
pwntools permette di interfacciarsi a processi eseguiti in locale o in remoto:

```
processo = process('mio_eseguibile')          # Avvia il processo 'mio eseguibile'
processo_remoto = remote('server.com', 4096) # Si collega al processo remoto su
                                              # server.com alla porta 4096
```

E' possibile anche richiamare direttamente GDB e interfacciarsi a un processo gia' aperto oppure farne aprire uno da GDB stesso:

```
gdb.attach(processo, 'break main\ncontinue')
processo_debug = gdb.debug('mio_eseguibile', 'break main\ncontinue')
```

ATTENZIONE!
Se si avvia un processo direttamente da GDB, esso inserisce di default alcune variabili nell'ambiente del processo ==> **GLI INDIRIZZI DELLE VARIABILI SULLO STACK DEL PROCESSO SONO DIVERSI DA QUELLI DEL PROCESSO ESEGUITO DA SHELL!**

pwntools: alcune funzioni utili (3/4)

Adesso che abbiamo aperto un canale di comunicazione con il processo come inviamo/riceviamo dati? pwntools mette a disposizioni diverse funzioni, alcuni esempi:

```
processo.sendline(“CIAO”)                    # Invia la stringa “CIAO”
dati_ricevuti = processo.recv(1024)          # Riceve fino a 1024 bytes
dati_ricevuti = processo.recvuntil(‘BBB’)    # Riceve finche’ non trova ‘BBB’
```

Si puo’ interagire manualmente con il processo trasformando la sessione in una sessione interattiva:

```
processo.interactive()
```

Una volta terminata l’interazione si puo’ chiudere la sessione con:

```
processo.close()
```

pwntools: alcune funzioni utili (4/4)

Adesso vorremmo inviare una sequenza di byte al processo al fine di eseguire un exploit. Supponiamo di voler inviare l'indirizzo di memoria:

0x7ffff7dd0d80

dovremmo convertirlo "a mano" in una stringa Python "little-endian" (se lavoriamo su CPU little-endian come le cpu Intel)

addr = '\x80\x0d\xdd\xf7\xff\xff\x00\x00'

tuttavia se abbiamo definito correttamente un contex bastera' eseguire:

addr = p64(0x7ffff7dd0d80)

e pwntool costruira' la giusta rappresentazione per noi occupandosi ad esempio di eventuali padding e dell'endianess! Ovviamente sono disponibili anche le funzioni **p32()** e **p16()**.

pwntools: tutto il resto

pwntools mette a disposizione moltissime funzioni utili, per maggiori dettagli

http://docs.pwntools.com/en/stable/

Fine :)