CyberSec

February 18, 2026

1 modulo fondamenti di programmazione

1.1 Corso Cybersecurity

---

2 Andrea Ribuoli

2.1 cheat-sheets

Nelle prime lezioni abbiamo approfondito la programmazione in **Python** avendo come obiettivo la comprensione pratica della strutturazione del codice e in particolare del ruolo svolto in Python dall'**indentazione** come mezzo per definire blocchi di codice e sotto blocchi.

Questa svolge un ruolo chiave nei cicli, nella definizione di funzioni, classi e metodi.

Ho presentato un esempio di foglio riassuntivo della sintassi Python utile per ripasso e approfondimento.

Un suggerimento pratico è stato quello di valorizzare due funzioni base, sempre disponibili, che sono `type()` e `dir()`. Io li chiamo gli strumenti per il **Python "orienteering"**.

- `type()` ci consente di interrogare il tipo (la classe) a cui la variabile indicata punta in questo momento
- `dir()` ci consente di listare i metodi che la classe dell'oggetto, e i suoi avi, ci mettono a disposizione

[2]:
```
def dirOnly(istanza):
    methods=dir(istanza)
    risultato = []
    for m in methods:
        if not m.startswith('_'):
            risultato.append(m)
    return (risultato)
```

[4]:
```
str(dirOnly([]))
```

[4]:
```
"['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']"
```

Ho adottato *Visual Studio Code* e in particolare il suo supporto degli **Jupyter Notebook** come mezzo per sperimentare l'uso del linguaggio analizzandolo *"chirurgicamente"*.

Lo strumento offre la possibilità di inserire tramite il linguaggio **MarkDown** commenti testuali.

Per la rapida condivisione degli elaborati individuali abbiamo adottato **GitHub** a cui gli allievi sono stati invitati ad iscriversi e creare un repository di nome **cybersec**.

Due argomenti sono stati trattati in forma teorica.

2.2 stack buffer overflow

Uno ha fornito le premesse per comprendere il concetto di **stack buffer overflow**. E' stato creato un semplice programma in **C** che utilizza la funzione C `gets()` che è da evitarsi in quanto consente ad un utente malintenzionato di portare facilmente in *crash* il programma.

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    char buffer[20];
    printf("Come ti chiami?\n");
    gets(buffer);
    printf("Benvenuto %s, iniziamo la lezione!\n", buffer);
}
```

[4]:
```
!gcc -o prova1 prova1.c
```

[5]:
```
!echo "Andrea" | prova1
```

```
Come ti chiami?
Benvenuto Andrea, iniziamo la lezione!
```

[6]:
```
!echo "Ora tento di passare al programma un contenuto eccedente il buffer" |␣
 ↪prova1
```

```
Come ti chiami?
Benvenuto Ora tento di passare al programma un contenuto eccecente il buffer,
iniziamo la lezione!
/QOpenSys/pkgs/bin/bash: line 1:  7458 Done                    echo "Ora tento
di passare al programma un contenuto eccecente il buffer"
      7459 Segmentation fault      (core dumped) | prova1
```

Si è verificato un **Segmentation fault**.

Una volte identificata la presenza di una tale falla di sicurezza un malintenzionato può costruire un particolare input che sovrascriva l'indirizzo di ritorno sullo stack in modo non casuale. L'obiettivo dell'hacker è fare in modo che il contenuto memorizzato dal programma C acquisendo l'input utente

che eccede la capacità del buffer sia puntato dal valore sovrascritto nell'indirizzo di ritorno. Quando il contenuto è casuale, si genera solitamente il segmentation fault (mostrato in precedenza) ma se il sistema operativo non limita la eseguibilità del codice eventualmente presente nella memoria dello stack, ecco che all'hacker viene offerta la possibilità di fare leva sulla falla di sicurezza descritta..

Per approfondire il tema esiste una buona pagina su Wikipedia

Da questa fonte prendiamo la seguente classificazione delle contromisure:

- A livello di **linguaggio**
- A livello di **codice sorgente**
- A livello di **compilatore**
- A livello di **sistema operativo**

Adottando un linguaggio come **Python** si previene il rischio evidenziato in **C**: non potranno esserci sconfinamenti in memoria, in quanto il controllo sulle dimesioni da allocare è automatico.

A livello di codice sorgente, ad esempio in C, può essere indagata la presenza di chiamate a funzioni che espongano a tali rischi e così manutenere il codice per eliminare le falle.

A livello di compilatore si può (anche in C) imporre che le verifiche che prevengono gli sconfinamenti siano inserite automaticamente: in tal caso accetto una perdita di prestazioni a favore della sicurezza (sempre senza dovere riscrivere i programmi sorgente).

A livello di sistema operativo sono state introdotte differenziazioni delle pagine di memoria così che *stack* e *heap* non siano **eseguibili**. Nel caso di una falla per la quale sia possibile sconfinare nel segmento di memoria dove risiede lo stack, sarà il sistema operativo a riconoscere che un determinato indirizzo non punta ad una parte di memoria eseguibile e quindi neutralizzare il potenziale attacco.

2.3 SQL injection

Il secondo tema trattato in forma teorica è la *SQL injection*. Questa ha a che fare con i cosiddetti siti *dinamici*. L'aggettivo "dinamico" è contrapposto al concetto di sito "statico". Il sito statico è quello in cui la navigazione è circoscritta all'attraversamento dei link iperstestuali offerti dalle pagine **HTML** che sono appunto statiche.

Un sito dinamico sfrutta spesso un database in cui sono memorizzate informazioni che vengono presentate all'utente secondo criteri di presentazione che l'utente stesso può indicare in campi di alcune **form** previste dal programmatore del sito.

Sfruttando le caratteristiche del protocollo *HTTP*, e sue estensioni, l'input dell'utente viene raccolto sul lato server da una vera e propria applicazione. Spesso questa applicazione è sviluppata in linguaggi di scripting quali il **PHP** che consente di essere incastonato all'interno di una struttura HTML.

La effettiva pagina HTML sarà restituita dopo la elaborazione delle sezioni con il PHP *embedded*. Il codice PHP può accedere alle informazioni raccolte dal browser a seguito dell'input dell'utente.

Se il codice PHP non è scritto attentamente, **pur risultando funzionale** dal punto di vista applicativo, potrebbe esporrre il sito ad attacchi.

Se infatti l'input utente viene utilizzato direttamente nel completamento di istruzioni SQL concatenando a parti di SQL le stringhe relative al valore di campi di selezione acquisiti dal browser ci si esponde ad un input malevolo.

Anche su questo tema troviamo del buon contenuto divulgativo su Wikipedia

Supponiamo di avere richiesto (e ricevuto) un nome utente nella variabile `userName`:

[6]:
```
userName = "Andrea"
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

Ora la variabile `statement` potra essere utilizzata per interagire con il database:

[7]:
```
statement
```

[7]:
```
"SELECT * FROM users WHERE name = 'Andrea';"
```

Il motore del database sarà in grado di restituire correttamente il dataset selezionato.

Il sito sta funzionando!

Tuttavia se non viene operata alcuna operazione di filtraggio sulla stringa passata dall'utente del sito può accadere questo:

[11]:
```
userName = "a';\nDROP TABLE departments;\nSELECT * FROM users WHERE name =␣
 ↪'Andrea"
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

[13]:
```
print(statement)
```

```
SELECT * FROM users WHERE name = 'a';
DROP TABLE departments;
SELECT * FROM users WHERE name = 'Andrea';
```

L'input malevolo, conscio del mancato controllo sulla variabile `userName` conduce ad un risultato disastroso per il sito!!

La pagina di Wikipedia citata menziona come alternativa al filtraggio l'adozione di una tipizzazione forte. Questa, in tutti i motori di database moderni, è implementata tramite i "PREPARED STATEMENTS"

La stringa contenente lo statement SQL presenta un segnaposto (o **placeholder**)

[15]:
```
statement = "SELECT * FROM users WHERE name = ?;"
```

Lo `statement` subirà prima una **PREPARE** e successivamente ci saranno una o più operazioni di **EXECUTE** con la indicazioni dei valori che risolvono i segnaposto. Il tentativo dell'input malevolo è neutralizzato: non potrà esssere modificata la logica di accesso al database.

Nelle prime giornate abbiamo introdotto la **programmazione socket** in Python essendo una conoscenza preliminare per affrontare il tema (assegnatomi nel programma) di scrittura di script per il **Penetration Testing**.

Si tratta di un tema complesso.

Tra l'altro in laboratorio i computer non consentivano di simulare la comunicazione tra computer perchè, per legittime ragioni di sicurezza, i tentativi di apertura di porte in ascolto richiedevano le password di amministratore dei computer che non erano a disposizione degli allievi.

Mi sono quindi limitato ad operare (e far eseguire gli esercizi) su **localhost**.

Quelli che seguono sono gli script Python finali del programma in ascolto e quello che simula la sessione utente elaborati incrementalmente durante le lezioni.

2.4 programma in ascolto

```
import socket
import pickle
server_socket = socket.socket()
host = '127.0.0.1'
port = 7654

try:
    f = open('utenti.dump', 'rb')
    try:
        utenti = pickle.load(f)
    finally:
        f.close()
except:
    utenti = {'admin':'èsegreta'}

server_socket.bind((host, port))
server_socket.listen(1)
for i in range(5):
    conn, addr_p = server_socket.accept()
    print(f"Connected by {addr_p}\n")
    conn.sendall(b'Indicami il tuo username: ')
    username = conn.recv(1024).decode()
    conn.sendall(b'Indicami la tua password: ')
    password = conn.recv(1024).decode()
    if username == 'admin' :
        if utenti[username] == password:
            conn.sendall(str(utenti).encode())
        else:
            conn.close()
            continue
    else:
        utenti[username] = password
    conn.close()
server_socket.close()
f = open('utenti.dump', 'wb')
pickle.dump(utenti, f)
```

```
f.close()
```

2.5 programma utente

```
import socket
s = socket.socket()
indirizzo = '127.0.0.1'
porta = 7654
s.connect((indirizzo, porta))
prompt = s.recv(1024)
username = input(prompt.decode())
s.sendall(username.encode())
prompt = s.recv(1024)
password = input(prompt.decode())
s.sendall(password.encode())
if username == 'admin':
    prompt=s.recv(1024)
    print(prompt.decode())
s.close()
```

2.6 uno Unix "password cracker"

In una lezione ho menzionato il libro The Cuckoo's Egg. È stato lo spunto per menzionare una funzione Python (oggi il modulo `crypt` è stato rimosso) che emulava il comportamento dell'algoritmo di crittografazione delle password nei sistemi Unix di qualche anno fa. Ho ritenuto utile parlarne e mostrarne il funzionamento.

[17]:
```
import crypt
hidden_password = crypt.crypt('Louvre')
```

[18]:
```
print(hidden_password)
```

```
Vhg4OqtZWMzfc
```

[19]:
```
hidden_password = crypt.crypt('Louvre')
print(hidden_password)
```

```
Yz.l5hIKZkXAg
```

La particolarità dell'algoritmo è che i primi due caratteri costituiscono il **seed**. Potendo leggere i primi due caratteri della password criptata siamo garantiti garantiti di poter replicare la generazione della cifratura nota la password in chiaro:

[20]:
```
print(crypt.crypt('Louvre', 'Vh'))
print(crypt.crypt('Louvre', 'Yz'))
```

```
Vhg4OqtZWMzfc
Yz.l5hIKZkXAg
```

Supponendo di avere avuto accesso al file utenti Unix con le password cifrate all'hacker era possibile verificare se qualche utente avesse adottato una password troppo semplice (di cui l'hacker si era in base alla sua esperienza costruito un elenco)

[22]:
```
print(hidden_password)
```

```
Yz.l5hIKZkXAg
```

[23]:
```
lista_password_banali = ['1234', 'casa', 'Laura', 'Gioconda', 'Louvre']
```

[26]:
```
for pw in lista_password_banali:
    hidden_password2 = crypt.crypt(pw, hidden_password[0:2] )
    if hidden_password2 == hidden_password:
        print("  utente sgamato!!  ")
```

```
 utente sgamato!!
```
