**Relazione Homework**

|  |  |
| --- | --- |
| **informazioni generali** |  |
| ***studente*** | Pizzi Andrea |
| ***e-mail*** | pizzi.199517@studenti.uniroma1.it |
| ***matricola*** | 1995517 |

**Sommario**

[Traccia 1 - Buffer Overflow 2](#_Toc168305823)

[Introduzione 2](#_Toc168305824)

[Step 0: Configurazione dell'ambiente di esecuzione 2](#_Toc168305825)

[Step 1: disabilitare le misure di sicurezza standard 2](#_Toc168305826)

[Step 2: Calcolare l'offset dello stack (posizione dello stack pointer RIP/ EIP in x32) 4](#_Toc168305827)

[Step 3: Assumere il controllo del RIP 5](#_Toc168305828)

[Step 4: Preparare la shellcode 5](#_Toc168305829)

[Step 5: Preparare la stringa malevola 8](#_Toc168305830)

[Step 6: Passare la stringa in input al programma senza inconvenienti di codifica 9](#_Toc168305831)

[Step 7: Gestire le variabili d'ambiente 9](#_Toc168305832)

[Step 8: Lancia una shell con permessi di root 11](#_Toc168305833)

[Risultati sperimentali: esempio di attacco 11](#_Toc168305834)

# Traccia 1 - Buffer Overflow

Realizzare un attacco di buffer overflow the permetta di aprire una shell su di un sistema target. A tale scopo si richiede sia di realizzare un programma vulnerabile ad un attacco di buffer overflow, sia di progettare e implementare la sequenza di byte che deve essere iniettata nel buffer per realizzare l’attacco. La shell deve essere eseguita con i privilegi del programma vulnerabile che viene sfruttato per il buffer overflow. (Opzionale) migliorare l’attacco provando a far guadagnare alla shell maggiori privilegi rispetto a quelli del programma di cui si è sfrutta la vulnerabilità.

# Introduzione

L'obiettivo del progetto è sfruttando una vulnerabilità nota di buffer overflow progettare un attacco shellcode in grado di aprire una shell con permessi di root. Il progetto si articola in una serie di script shell, python e codice C per la simulazione del programma vulnerabile e per l'esecuzione dell'attacco. Nelle sezioni seguenti verrà spiegata la struttura del progetto ed i passi intrapresi per la sua realizzazione.

# Step 0: Configurazione dell'ambiente di esecuzione

L'intero ambiente di esecuzione dell'attacco è virtualizzato tramite container gestiti da un sistema [podman](https://podman.io/).

* Container docker configurato con sistema operativo **Debian 12 Bookworm** personalizzato con i seguenti tools
* gdb ( più estensione gdb-peda)
* editor nano python3
* Programma compilato con GCC 6.2
* Struttura dell'homework (albero delle directory)
  + **assembly**: codice assembly per shellcode
  + **bin**: programmi compilati (eseguibili)
  + **data**: script e file necessari per l'attacco
  + **obj**: File oggetto
  + **src**: Sorgenti dei programmi compilati
  + **./**: Dockerfile e script per costruire e lanciare l'ambiente conteinerizzato

# Step 1: disabilitare le misure di sicurezza standard

Disabilitare la randomizzazione degli indirizzi (per il sistema host)

sysctl kernel/randomize\_va\_space

Disabilitare i meccanismi di protezione in fase di compilazione

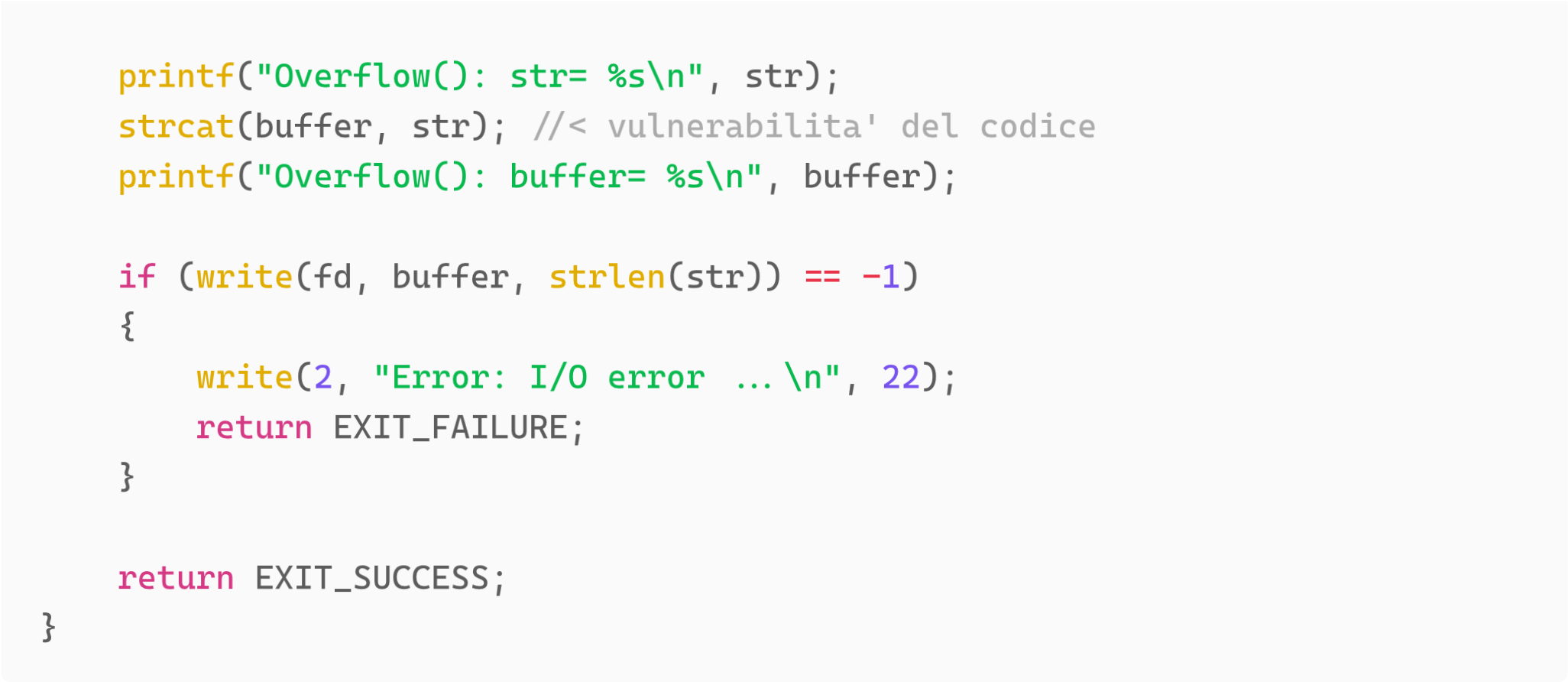
-z execstack : permetti l'esecuzione di codice arbitrario nello stack

-fno-stack-protector : disabilita i meccanismi di protezione dello stack

Il programma C esempio sarà una funzione in grado di leggere da stdin una stringa, bufferizzarla e successivamente scriverla su un file.

La funzione C in analisi





Per la compilazione della shellcode invece è stato realizzato il seguente makefile



La vulnerabilità del programma risiede principalmente nel fatto che non viene effettuano nessun controllo sulla dimensione della stringa che viene scritta nel buffer pertanto qualsiasi stringa di lunghezza maggiore di 128 byte provocherà l'overflow.

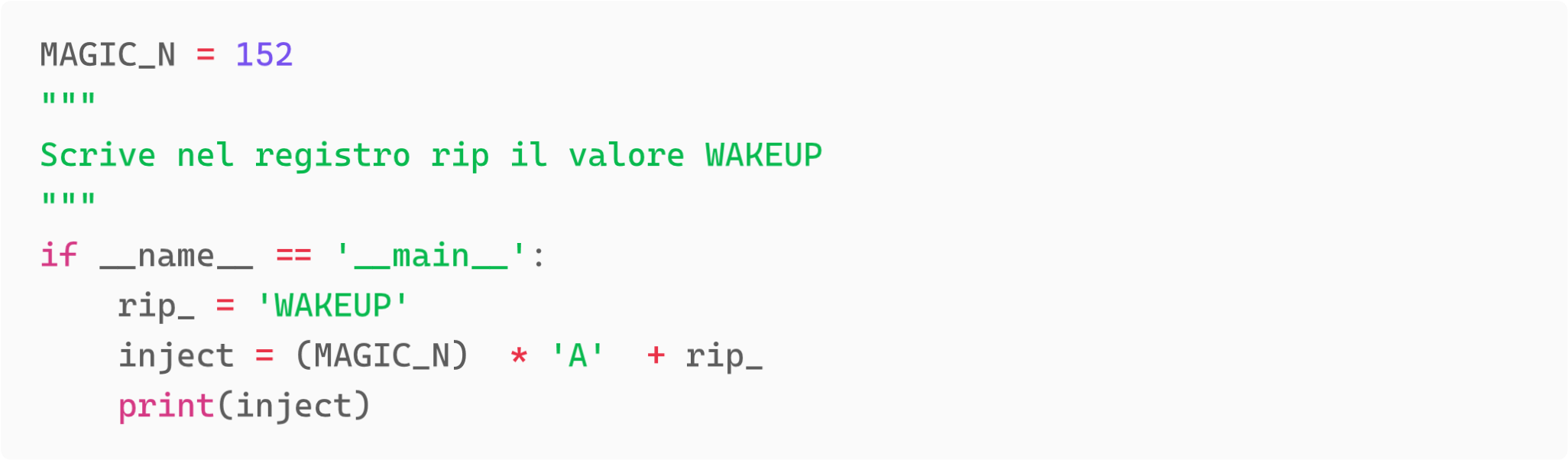
# Step 2: Calcolare l'offset dello stack (posizione dello stack pointer RIP/ EIP in x32)

Per l'analisi viene utilizzata un’estensione del tradizionale debbuger C gdb con funzionalità avanzate di exploit **gdb-peda**

1. Tramite il comando del debugger pattern create viene generato un file contenente una sequenza di valori generati secondo un pattern di dimensione maggiore di quella prevista del buffer
2. Si imposta il debugger per tracciare l'esecuzione del programma target
3. Si immette come input al programma infetto il file pattern generato (vedere pattern.txt)
4. Tramite il comando pattern search viene estratto offset dello stack pointer (nel caso corrente pari a 152 byte) che indica il numero di byte che separano il valore del RIP dalla cima dello stack.

# Step 3: Assumere il controllo del RIP

Per testare il controllo sul contenuto del registro RIP è stato realizzato uno script python *(versione python3)* che genera la stringa da somministrare di dimensione offset + 6. Il test permette di verificare il controllo dei valori nel registro RIP

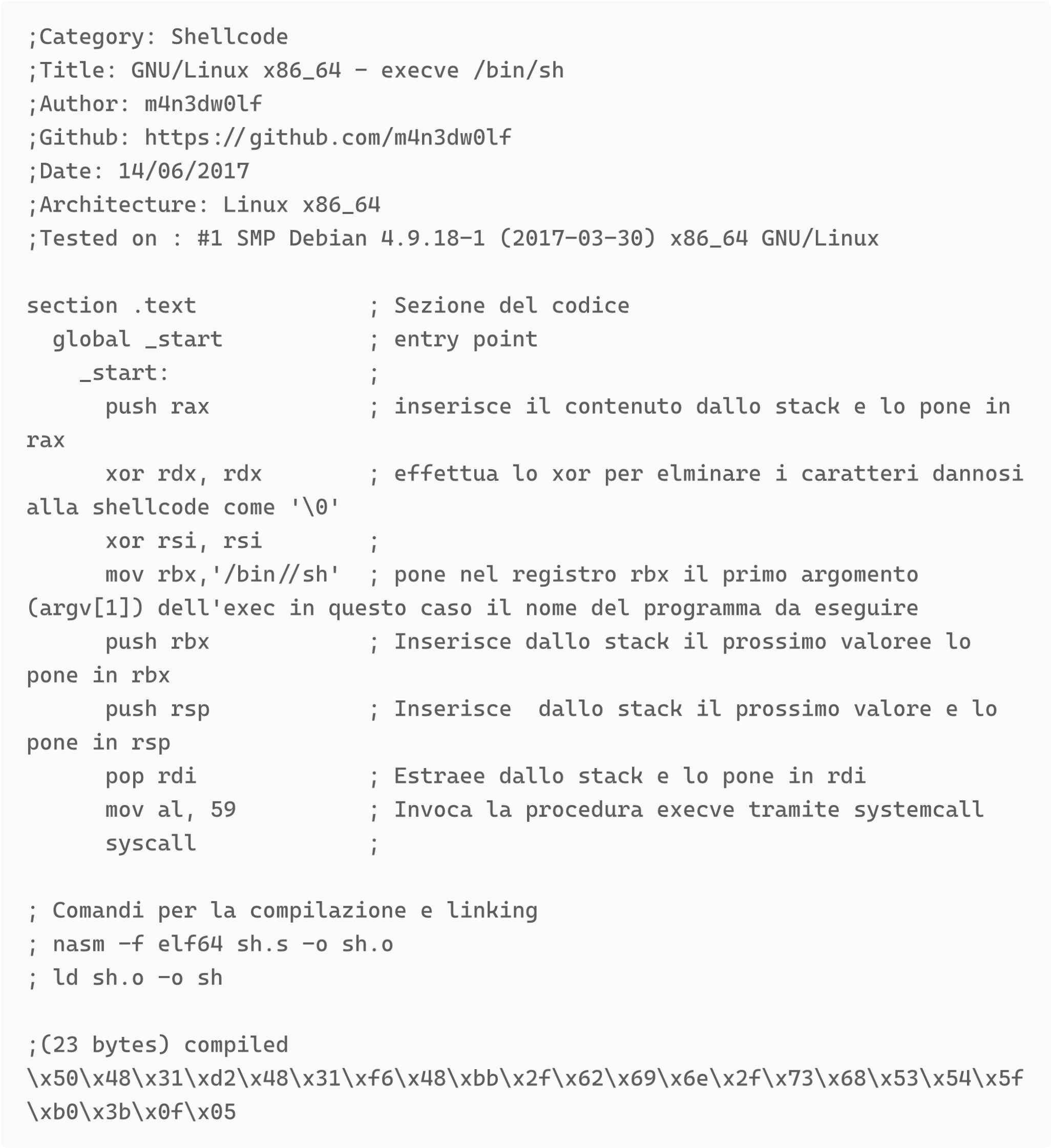


# Step 4: Preparare la shellcode

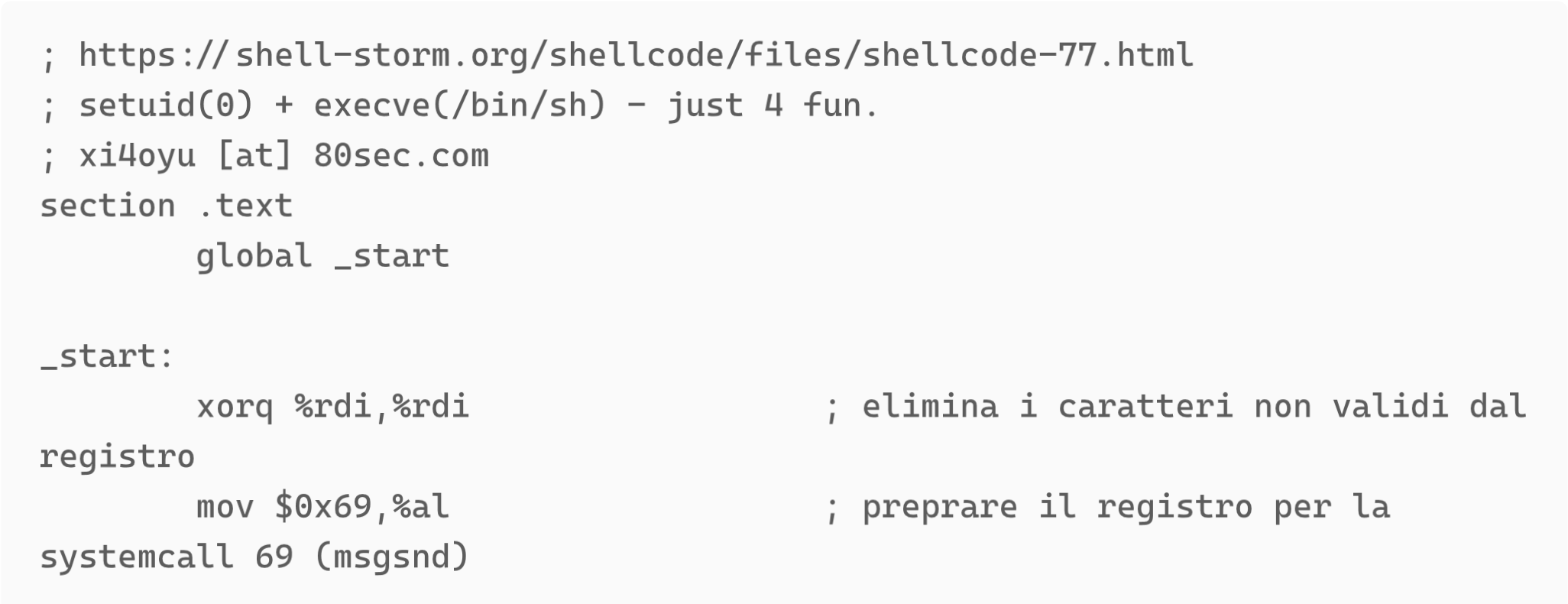
Per la preparazione del codice della shellcode è necessario in primis realizzare il relativo programma in assembly (notare che un attacco ha una bassa portabilità e la sua riuscita dipenderà anche dall'architettura del processore) in base al processore della macchina da attaccare. In questo caso l'attacco sarà eseguito su una macchina con architettura AMD64.

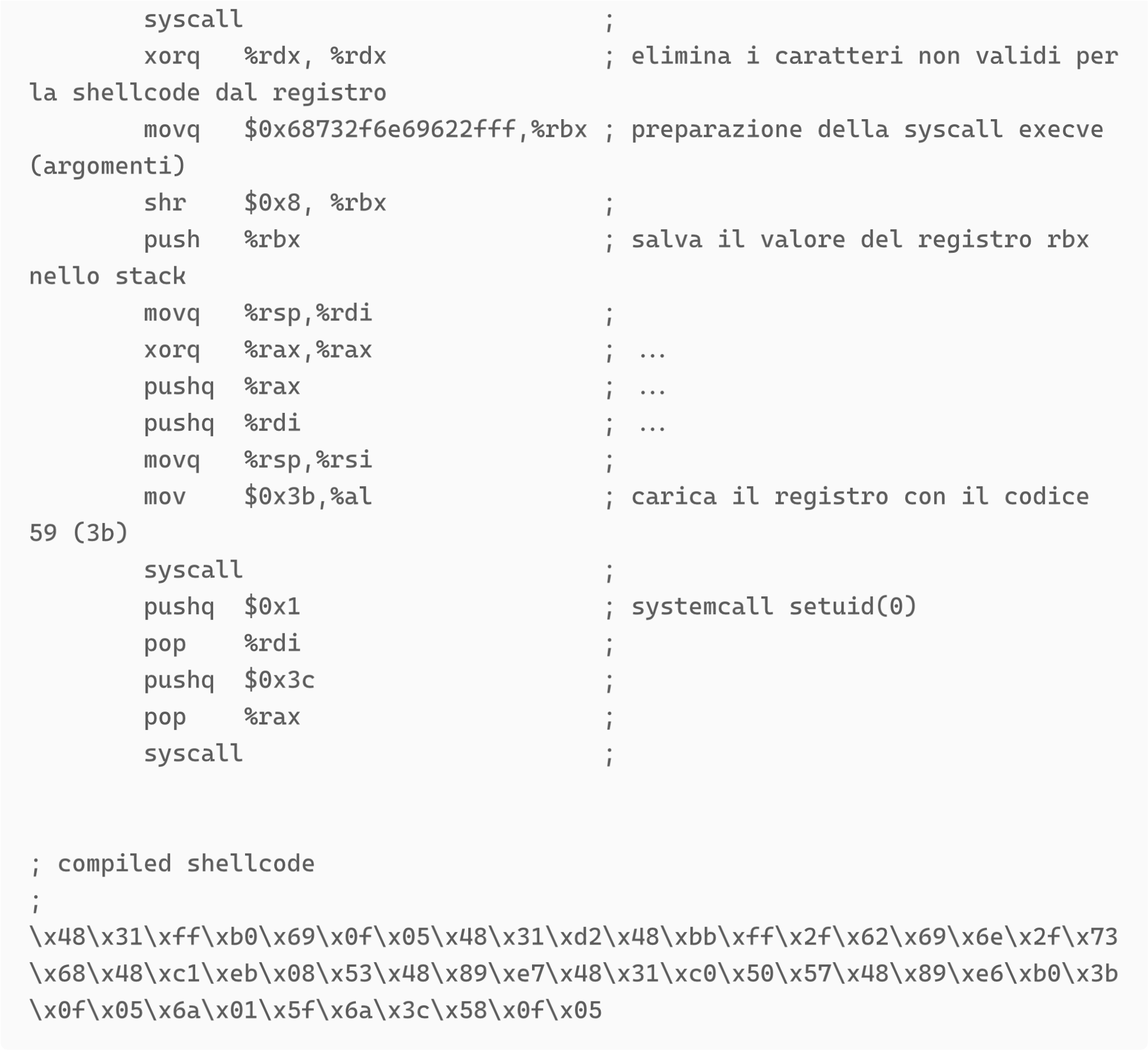
Per ragioni di praticità il codice della shellcode è stato recuperato in rete dove si possono trovare esempi di shellcode già pronte e compilate. Di seguito vengono riportati i codici assembly con i relativi riferimenti.

La seguente shellcode lancia una nuova shell tramite la chiamata alla funzione execve (systemcall 59).



codice assembly che effettua l'operazione di setuid(0) per acquisire privilegi di root e quindi lanciare una shell con gli stessi privilegi





La stringa esadecimale già fornita come risorsa può essere ottenuta compilando il codice assembly ed analizzandolo con un qualsiasi editor esadecimale altrimenti si può fare ricorso all'utility di sistema **objdump**

# Step 5: Preparare la stringa malevola

Calcolato la stringa per la shellcode abbiamo ottenuto quasi tutti gli elementi necessari per creare la stringa da passare al programma perché la shellcode venga eseguita. Attualmente disponiamo delle seguenti informazioni

* Offset del registro RIP rispetto alla cima dello stack (152 byte)
* Shelcode in valori esadecimali da eseguire

Non rimane che individuare l'indirizzo di memoria con cui sostituire il valore del registro RIP in modo tale che la shellcode venga effettivamente eseguita per fare ciò prepariamo un apposito input del programma. La stringa che sarà costruita per lo scopo può essere suddivisa nelle seguenti parti



Il valore RIP sarà scelto a caso l'interesse effettivo attualmente è sulla parte iniziale della stringa Quest’ultima è fondamentale perché la shellcode venga eseguita correttamente in quanto serve a riservare un pool di indirizzi di memoria contigui che verranno utilizzati come possibili valori da inserire nel RIP.

Più nel dettaglio una volta avvenuto overflow del programma in esecuzione il processore continuerà a processare i valori inseriti nello stack fino al byte 152 rispetto all'indirizzo puntato dallo stack-pointer $eip. I byte seguenti verranno interpretati come indirizzi di ritorno della procedura. Sostituendo accuratamente tali byte il IC effettuerà un salto all'indietro nell'area di valori nop . Processati tutti i valori nop al termine dell'area si troverà il codice della shellcode che verrà caricato correttamente dall'inizio e sarà eseguito come un nuovo programma.

La scelta dell'indirizzo tra quelli occupati dal blocco di valori di nop deve essere effettuata con criterio cercando di prendere l'indirizzo più centrale del blocco per essere sicuri che tra un’esecuzione all'altra la shellcode funzioni sempre. Scegliendo infatti indirizzi troppo vicini all'inizio o alla fine del blocco si rischia, a causa di altri valori che si possono trovare nello stack e alla randomicità nell'assegnazione dell’area di memoria, di 'mancare' l'indirizzo utilizzato per il salto e di non trovare nella posizione il valore nop. Generalmente si consiglia di selezionare un'area abbastanza ampia di valori nop per facilitare l'operazione di scelta dell'indirizzo, in questo caso la dimensione utilizzata è stata di 64 byte.

Scelto quindi l'indirizzo a cui saltare basta sostituire il valore RIP casuale con l'indirizzo scelto e la stringa malevola è pronta per poter essere utilizzata.

La procedura per l'homework è stata eseguito tramite **gdb-peda** e uso di script python. Tramite l'utility di ispezione della memoria di gdb-peda sono stati individuati gli indirizzi mentre tramite lo script python è stata costruita la stringa mostrata all'inizio.

Lo script python



# Step 6: Passare la stringa in input al programma senza inconvenienti di codifica

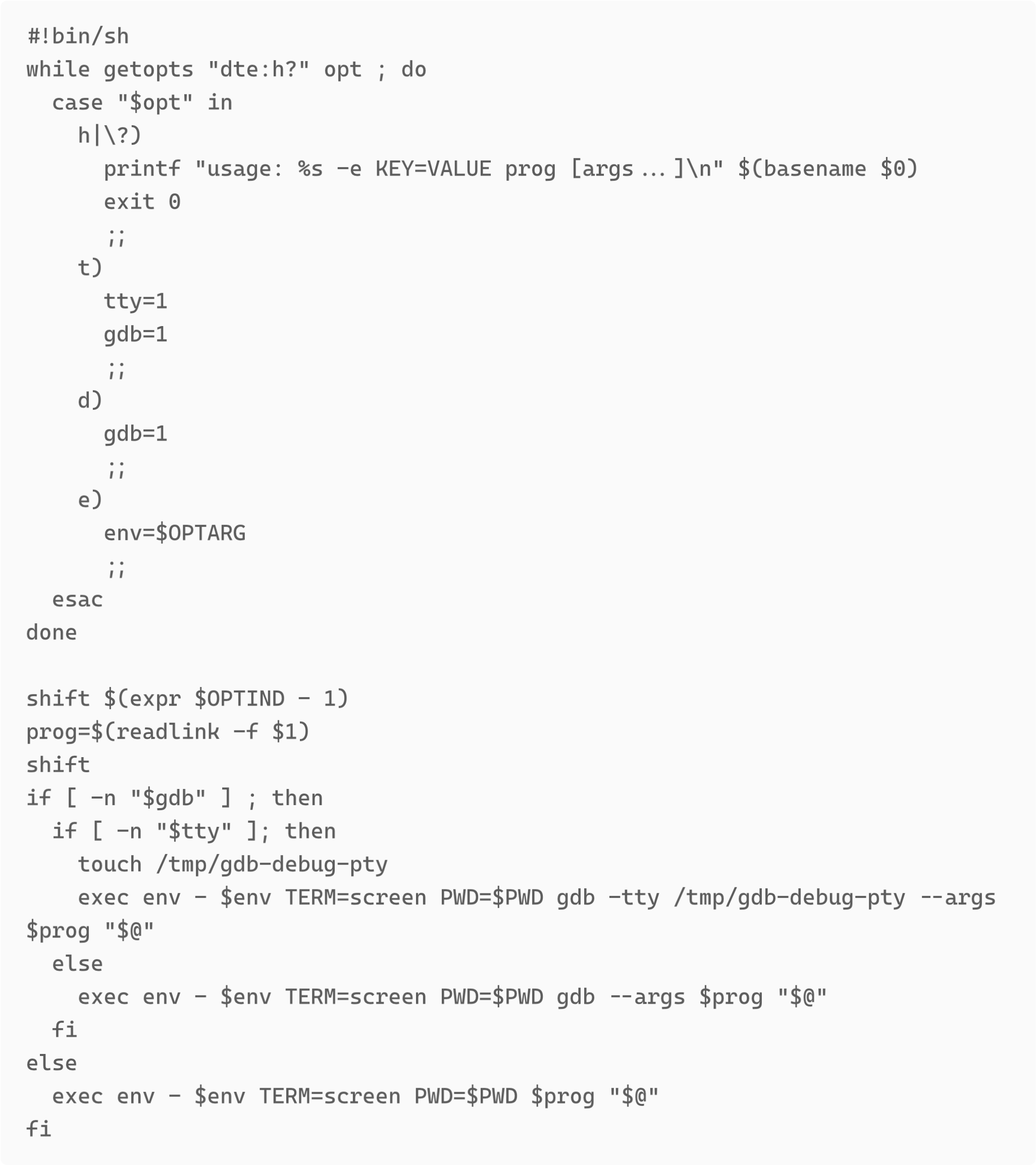
Fino alla versione di python2 la gestione delle stringhe non che la loro rappresentazione era effettuata in modo basilare senza codifiche da python3 la gestione delle stringhe è molto più organica convertendo i caratteri in oggetti stringa a cui viene associata una codifica, tutto ciò sarà meno utile per i nostri scopi. Applicando una codifica sui caratteri alcuni caratteri effettivi potrebbero essere sostituiti oppure eliminati alterando così la shellcode progettata. Per evitare tale inconveniente viene adottata una soluzione alternativa all'uso dell'instruzione print. Prima di tutto i caratteri manipolati vengono utilizzato solamente in formato binario. Dopo di che la stringa non viene visualizzati su standard di input/output ma copiata in un file binario. In tal modo i dati vengono gestiti unicamente come insieme di byte senza inconvenienti di codifica.

# Step 7: Gestire le variabili d'ambiente

Un altro ostacolo tecnico alla shellcode sono la presenza di variabili d'ambiente nei diversi ambienti di esecuzione (shell, gdb etc.). Quando viene eseguito un qualsiasi programma il sistema inserisce di automatico in cima allo stack i valori relativi alle variabili d'ambiente ciò provoca lo shift delle istruzioni nello stack alterando così l'efficacia della shellcode progettata. Per evitare tali inconvenienti è necessario trovare una soluzione per rendere l'esecuzione della shell code indipendente dalla presenza o meno di variabili d'ambiente. Una soluzione rapido ed efficace è l'uso di uno script wrapper che modifica opportunamente le variabili d'ambiente perché la shellcode possa funzionare. La funzione wrapper utilizzata è uno script sh preso dal seguente post sul sito [StackOverflow](https://stackoverflow.com/)

[Buffer overflow works in gdb but not without it.](https://stackoverflow.com/questions/17775186/buffer-overflow-works-in-gdb-but-not-without-it)

Script sh utilizzato



# Step 8: Lancia una shell con permessi di root

Abbiamo ottenuto tutti gli elementi per eseguire l'attacco di shellcode non rimane che lanciarlo

1. Si genera tramite lo script inject.py il file contenente la stringa da passare al programma vulnerabile
2. L'attacco viene lanciato tramite il seguente script sh(load.sh) che carica la stringa e la somministra al programma tramite il programma wrapper wrap.sh.

shell

c

ode

=

$(

ca

t

inje

c

t

)

ba

sh

wr

a

p

.

sh

m

a

in

$

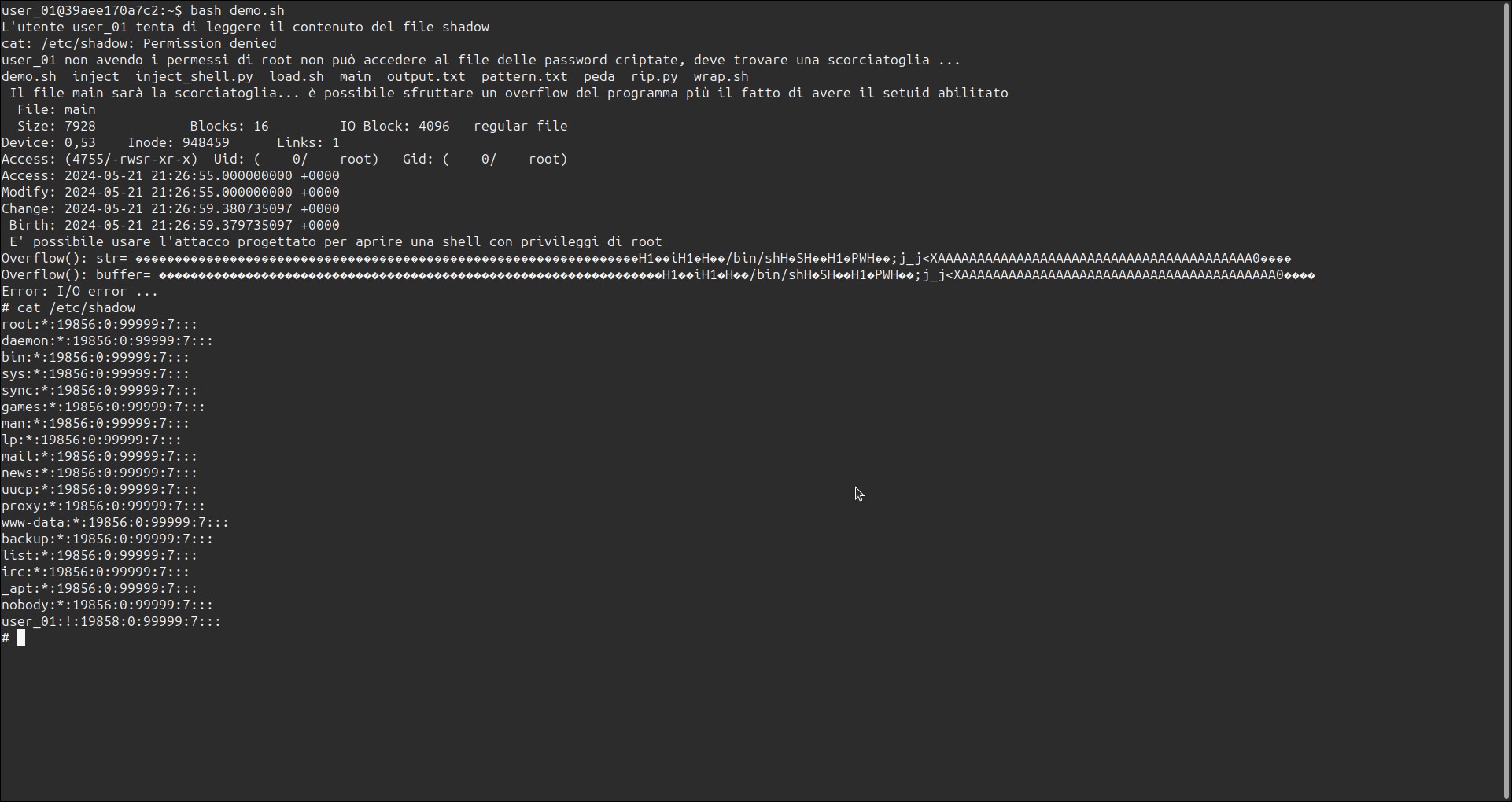
shell

c

ode

1. Viene aperta una nuova shell con i permessi di setuid previsti (il file appartiene al root quindi il programma verrà eseguito con privilegi di root).

# Risultati sperimentali: esempio di attacco



**notare che nell'esempio mostrato non viene mai chiesto all'utente di inserire una password neanche per lanciare una shell come root. Notare inoltre che l'utente user\_01 non potrebbe neanche diventare root non essendo un utente sudoers**

Lo screenshoot mostra come dovrebbe apparire l'attacco. Le conseguenze dell'apertura di una shell come root sono abbastanza gravi permettendo all'attaccante di

prendere il controllo del sistema modificando le password degli utenti root o rimuovendoli

accedere deliberatamente a tutti i file del sistema (sia in lettura che scrittura)

Installare malware come backdoor, keylogger etc... per continuare l'attività in incognito

Nell'immagine presentata vediamo come si potrebbe realizzare l'attacco al punto due della lista accedendo al file *shadow*.

L'attacco come descritto si configura come intrusione o accesso non autorizzato al sistema. Una possibile ulteriore linea di difesa successiva all'intrusione potrebbe essere un HIDS (Host-based intrusion detection system) in grado di rilevare l'intrusione in base al comportamento anomalo del programma.