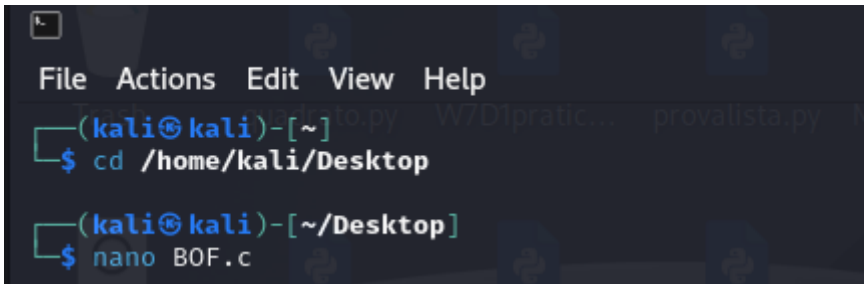


W17D4 – buffer overflow

Esercizio obbligatorio

Ho iniziato a scrivere il codice volutamente vulnerabile al buffer overflow seguendo le indicazioni riportate nella consegna dell'esercizio, scrivendo poi il codice in un file chiamato "BOF.c".



```
(kali㉿kali)-[~]  
$ cd /home/kali/Desktop  
  
(kali㉿kali)-[~/Desktop]  
$ nano BOF.c
```

Ho riportato con attenzione il codice vulnerabile al buffer overflow, salvando poi il file.



```
kali@kali: ~/Desktop  
File Actions Edit View Help  
GNU nano 8.0 BOF.c  
#include <stdio.h>  
  
int main () {  
    char buffer [10];  
  
    printf ("Si prega di inserire il nome utente:");  
    scanf ("%s", buffer);  
  
    printf ("Nome utente inserito: %s\n", buffer);  
  
    return 0;  
}
```

A questo punto ho compilato il file utilizzando il comando `gcc -g BOF.c -o BOF` ed ho testato la vulnerabilità del buffer overflow eseguendo il programma con `./BOF`. Nel primo caso, ho inserito un nome utente di 6 caratteri, inferiore al limite di 10, ed è stato salvato correttamente senza problemi. Nel secondo caso ho invece inserito un numero di caratteri ben superiore al limite di 10 ed è stato riportato un segmentation fault, l'errore di segmentazione, che si verifica quando si tenta di scrivere in una porzione di memoria destinata ad altro.

```
(kali㉿kali)-[~/Desktop]
$ gcc -g BOF.c -o BOF

(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:MariaZ
Nome utente inserito: MariaZ

(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:adsfertyufidsguuddsigsfmbncxbshdfgkjitodandhf
Nome utente inserito: adsfertyufidsguuddsigsfmbncxbshdfgkjitodandhf
zsh: segmentation fault ./BOF
```

Come suggerito nell'esercizio, ho modificato il codice aumentando la dimensione del vettore a 30.



```
kali@kali: ~/Desktop
File Actions Edit View Help
GNU nano 8.0 BOF.c
#include <stdio.h>

int main () {
char buffer [30];

printf ("Si prega di inserire il nome utente:");
scanf ("%s", buffer);

printf ("Nome utente inserito: %s\n", buffer);

return 0;
}
```

Ho di nuovo compilato il programma con `gcc -g BOF.c -o BOF` e l'ho eseguito per testarlo. Nel primo caso, ho inserito un nome utente di 26 caratteri, inferiore al limite di 30 e il nome utente è stato salvato correttamente. Nel secondo caso ho invece riprodotto l'errore di segmentazione dando un input con un numero di caratteri di gran lunga superiore al limite del buffer. È stato quindi riportato nuovamente un segmentation fault. Questo significa che, come immaginavo, aumentare la dimensione del vettore non è una misura sufficiente ad eliminare il rischio di un buffer overflow, perché ci potrebbe essere un codice malevolo molto lungo che supera il limite e crea dei danni. A questo punto è necessario introdurre ulteriori controlli di sicurezza e sanitizzare l'input utente.

```
(kali㉿kali)-[~/Desktop]
$ gcc -g BOF.c -o BOF

(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:MariaZanchettaElenaMassimo
Nome utente inserito: MariaZanchettaElenaMassimo

(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:MariaZanchettaElenaMassimoManuelValerioChiaraPaoloLucaVittoriaSimoneDanieleTommasoGiuseppe
Nome utente inserito: MariaZanchettaElenaMassimoManuelValerioChiaraPaoloLucaVittoriaSimoneDanieleTommasoGiuseppe
zsh: segmentation fault ./BOF
```

Esercizio facoltativo.

Per risolvere il problema del buffer overflow, è dunque necessario implementare dei controlli di sicurezza aggiuntivi e sanitizzare l'input utente. La funzione `fgets` ha una funzione importantissima in questo, perché permette di impostare il numero massimo di caratteri da leggere, evitando che l'input utente superi il limite del buffer e prevenendo così il buffer overflow e la sovrascrittura di altre aree di memoria. Viene limitata la lunghezza dell'input, indicando esplicitamente il numero massimo di caratteri da leggere e si implementa così un controllo che `scanf` non consente. Usare funzioni più moderne come `scanf_s` oppure `snprintf` aiuta a proteggere in modo ancora più accurato, perché esse sono la versione più sicura di `scanf` e `sprintf`. Con `sizeof(buffer)` viene indicata la lunghezza massima del buffer da non superare, rendendo `scanf_s` più sicura di `scanf`, mentre `snprintf` salva il risultato della stringa nel buffer (come `sprintf`) ma inserisce un ulteriore controllo riguardo al massimo numero di caratteri da scrivere nella porzione di memoria allocata, compreso il null terminator. Questo avviene grazie a `sizeof(buffer)`, che evita che si cerchi di scrivere oltre lo spazio di memoria allocato, prevenendo quindi il buffer overflow. Si possono aggiungere anche controlli riguardo alla validazione dell'input, evitando che caratteri speciali o comunque non utili allo scopo della stringa da inserire vengano dati come input, magari per una SQL injection o per altri fini malevoli. Nel caso dell'esercizio il nome utente può contenere anche caratteri speciali oppure dei numeri, però se si fosse trattato di inserire un nome e un cognome, allora sarebbe stato opportuno accertarsi che venissero inserite solo delle lettere e non dei numeri o dei caratteri speciali. Questo può essere fatto sempre con le funzioni già note oppure inserendo delle regular expressions, che aiutano ad individuare dei path accettabili, per esempio se bisogna inserire un indirizzo email, distinguendo i path corretti dalle stringhe che possono nascondere un fine malevolo. È possibile anche inserire delle limitazioni a certi caratteri usati spesso in un attacco come SQLi, per esempio `|` oppure `&`. Si possono anche inserire delle protezioni avanzate come stack canaries (`-fstack-protector`), che aiutano a prevenire un buffer overflow perché controlla il valore del canary, che è un valore segreto inserito nello stack tra le variabili locali e l'indirizzo di ritorno di una funzione. Se ci si accorge che il canary è stato alterato, viene bloccata subito l'esecuzione del codice potenzialmente malevolo, proteggendo dalle conseguenze della sua esecuzione. Inoltre è possibile abilitare il flag `-D_FORTIFY_SOURCE=2` a protezione del compilatore, che significa che viene abilitato un livello di sicurezza più robusto, pari a 2, per controllare che non avvenga il buffer overflow e che non vi siano altri tentativi malevoli manipolando la memoria con C.

Alla luce di queste considerazioni, ho provato a modificare il codice iniziale, usando `fgets` per limitare il numero di caratteri da leggere ed inserendo un livello di protezione aggiuntivo per il compilatore. Ho creato un nuovo file chiamato `BOFsecure.c` ed ho scritto il codice modificato.

```
kali@kali: ~/Desktop
File Actions Edit View Help
GNU nano 8.0 BOfsecure.c
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[10];

    printf("Si prega di inserire il nome utente (max 9 caratteri):\n");

    // fgets e non scanf per limitare la lunghezza input e prevenire buffer overflow
    fgets(buffer, sizeof(buffer), stdin);

    // rimuove il carattere enter o a capo dall'input se presente
    buffer[strcspn(buffer, "\n")] = 0;

    printf("Nome utente inserito: %s\n", buffer);

    return 0;
}
```

Ho quindi testato lo script, compilandolo con misure di sicurezza più stringenti e poi eseguendo il programma. Nel primo caso, ho dato un input pari a 9 caratteri che è stato correttamente inserito e salvato, dato che rispettava il limite massimo di 9 caratteri. Nel secondo caso, ho inserito un input di 13 caratteri, superiore al limite del buffer e i controlli di sicurezza aggiuntivi hanno funzionato. L'input è stato sanitizzato, rispettando il limite massimo di caratteri ed evitando che venissero sovrascritte aree di memoria non accessibili, magari inserendo del codice malevolo e modificando il flusso di esecuzione del programma. Con i controlli aggiuntivi del compilatore, non sarebbe stato eseguito un codice superiore alle dimensioni del buffer. Come input del nome utente sono stati salvati solamente i 9 caratteri massimi, rimanendo all'interno dello spazio allocato per il buffer, mentre gli altri caratteri in più non sono stati sovrascritti da nessuna parte. Ora l'input viene sanitizzato e controllato nella sua lunghezza affinché non si scriva in porzioni di memoria destinate ad altro, eliminando il problema del buffer overflow e rendendo il codice più sicuro.

```
(kali@kali)-[~/Desktop]
$ gcc -o BOfsecure BOfsecure.c -fstack-protector -D_FORTIFY_SOURCE=2

(kali@kali)-[~/Desktop]
$ ./BOFsecure
Si prega di inserire il nome utente (max 9 caratteri):
GiuseppeT
Nome utente inserito: GiuseppeT

(kali@kali)-[~/Desktop]
$ ./BOFsecure
Si prega di inserire il nome utente (max 9 caratteri):
GiuseppeMario
Nome utente inserito: GiuseppeM
```