

INDICE

INDICE.....	1
INTRODUZIONE ED OBIETTIVO.....	2
METODOLOGIA OPERATIVA.....	3-6
CONCLUSIONE.....	7

INTRODUZIONE ED OBIETTIVO

INTRODUZIONE: il laboratorio consiste nell'analisi di un semplice programma scritto in linguaggio C progettato per leggere un input da tastiera e stamparlo a video. Attraverso la modifica controllata del codice e l'inserimento di input di lunghezza variabile, viene osservato il comportamento del programma in presenza di una gestione non sicura della memoria.

OBIETTIVO: riprodurre un errore di segmentazione causato da un buffer overflow e verificare se l'aumento della dimensione del buffer sia sufficiente a eliminare la vulnerabilità, analizzando le cause tecniche del problema.

METODOLOGIA OPERATIVA

```
(root㉿kali)-[~/home/kali/Desktop]
# cd ~/home/kali/Desktop

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

KALI episode [In esecuzione] - Oracle VirtualBox : 1

File Macchina Visualizza Inserimento Dispositivi Aiuto

Session Actions Edit View Help

GNU nano 8.7 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;
}
```

L'attività ha avuto inizio con la creazione del file sorgente contenente il codice del programma.

Utilizzando l'editor di testo presente nell'ambiente Kali Linux, è stato predisposto un nuovo file in linguaggio C, destinato a ospitare l'implementazione del programma oggetto dell'analisi.

All'interno del file è stato inserito il codice che definisce la funzione principale e un buffer di dimensione fissa, utilizzato per memorizzare l'input fornito dall'utente; questa fase ha consentito di predisporre l'ambiente di lavoro necessario per le successive operazioni di compilazione ed esecuzione, mantenendo una struttura semplice e focalizzata sull'analisi del comportamento del programma in presenza di input non controllato.

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

(root㉿kali)-[~/home/kali/Desktop]
# ls -l BOF
```

```
-rwxr-xr-x 1 root root 17272 Feb  5 17:16 BOF
```

Una volta completata la scrittura del codice sorgente, il file è stato compilato correttamente, generando il relativo file eseguibile, la compilazione è avvenuta senza errori, producendo un binario pronto per l'esecuzione.

La presenza dell'eseguibile e i relativi permessi di esecuzione confermano che il codice sorgente è stato tradotto correttamente e che il programma è pronto per essere avviato per le successive fasi di test e analisi del comportamento.

```
[root@kali]~/Desktop]
# ./BOF

Si prega di inserire il nome utente: ajeje
Nome utente inserito: ajeje

[root@kali]~/Desktop]
# ./BOF

Si prega di inserire il nome utente: hfsdhfiebafbalcdahdiahfaofeaioaifhiad
Nome utente inserito: hfsdhfiebafbalcdahdiahfaofeaioaifhiad
zsh: segmentation fault ./BOF
```

Durante l'esecuzione del programma con un input di lunghezza contenuta, il valore inserito viene correttamente memorizzato all'interno del buffer e successivamente stampato a video, senza generare anomalie.

In questo caso, la quantità di dati fornita dall'utente rientra nei limiti della memoria allocata, consentendo al programma di terminare correttamente.

Al contrario, quando viene inserita una stringa significativamente più lunga, il programma va incontro a un errore di segmentazione: questo comportamento è dovuto al fatto che l'input supera la capacità del buffer definito nel codice, causando una scrittura oltre i limiti della memoria allocata.

La conseguente corruzione della memoria porta al crash del programma, evidenziando la presenza di una vulnerabilità di tipo buffer overflow.

```
GNU nano 8.7
#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;
}
```

In questa fase il codice sorgente è stato modificato aumentando la dimensione del buffer destinato a memorizzare l'input dell'utente.

La dimensione del vettore è stata portata da 10 a 30 caratteri, con l'obiettivo di verificare se un'area di memoria più ampia fosse sufficiente a evitare il verificarsi dell'errore di segmentazione osservato in precedenza; ma la logica del programma e la modalità di acquisizione dell'input sono rimaste invariate, consentendo di isolare l'effetto della sola modifica della dimensione del buffer sul comportamento dell'applicazione.

```
[root@kali]~/Desktop]
# ./BOF

Si prega di inserire il nome utente: ajeje
Nome utente inserito: ajeje

[root@kali]~/Desktop]
# ./BOF

Si prega di inserire il nome utente: qwertyuiopasdfghjklzxcvbnmqwer
Nome utente inserito: qwertyuiopasdfghjklzxcvbnmqwer
zsh: segmentation fault ./BOF
```

Dopo l'aumento della dimensione del buffer a 30 caratteri, il programma è stato nuovamente eseguito con input di diversa lunghezza.

Con un input contenuto, il programma continua a funzionare correttamente, memorizzando e stampando il valore inserito senza generare errori; tuttavia, inserendo una stringa più lunga della dimensione del buffer, si verifica nuovamente un errore di segmentazione. Questo risultato conferma che l'aumento della dimensione del buffer non elimina la vulnerabilità, poiché la funzione utilizzata per leggere l'input non impone alcun limite sulla quantità di dati acquisiti, di conseguenza l'input eccedente continua a causare una scrittura oltre i limiti della memoria allocata, rendendo il programma ancora vulnerabile a un buffer overflow.

```
#include <stdio.h>

int main() {
    char buffer[30];
    printf("Si prega di inserire il nome utente: ");
    fgets(buffer, sizeof(buffer), stdin);

    printf("Nome utente inserito: %s\n", buffer);
    return 0;
}
```

In questa fase il codice è stato modificato intervenendo sulla modalità di acquisizione dell'input dell'utente.

La funzione precedentemente utilizzata, che non prevedeva alcun controllo sulla lunghezza della stringa inserita, è stata sostituita con una funzione che limita automaticamente il numero di caratteri letti in base alla dimensione reale del buffer; in questo modo, l'input non può più eccedere lo spazio di memoria allocato, impedendo la scrittura oltre i limiti del buffer. A seguito di questa modifica, l'esecuzione del programma non genera più errori di segmentazione, anche in presenza di input di lunghezza elevata, dimostrando l'efficacia della mitigazione applicata.

```
[root@kali]~[/home/kali/Desktop]
# ./BOF

Si prega di inserire il nome utente: qwertyuiopasdfghjklzxcvbnmqwer
Nome utente inserito: qwertyuiopasdfghjklzxcvbnmqwe
```

Dopo la modifica della funzione di acquisizione dell'input, il programma è stato nuovamente eseguito inserendo una stringa di lunghezza superiore alla dimensione del buffer; in questo caso, il programma non genera alcun errore di segmentazione e continua a funzionare correttamente.

L'input viene automaticamente limitato alla dimensione massima consentita dal buffer, evitando la scrittura oltre i limiti della memoria allocata.

Questo comportamento conferma che la vulnerabilità di tipo buffer overflow è stata correttamente mitigata attraverso il controllo esplicito dell'input.

CONCLUSIONE

L'attività svolta ha consentito di analizzare in modo pratico il comportamento di un programma in linguaggio C in presenza di una gestione non sicura dell'input, evidenziando come un buffer overflow possa verificarsi anche in applicazioni molto semplici.

Attraverso l'esecuzione controllata del programma con input di diversa lunghezza, è stato possibile osservare il verificarsi di un errore di segmentazione causato dalla scrittura oltre i limiti della memoria allocata.

L'esercizio ha inoltre dimostrato che l'aumento della dimensione del buffer, pur riducendo temporaneamente la probabilità di errore, non elimina la vulnerabilità se la funzione utilizzata per acquisire l'input non impone alcun limite sulla quantità di dati letti.

Solo intervenendo sulla modalità di gestione dell'input è stato possibile prevenire in modo efficace il buffer overflow, garantendo il corretto funzionamento del programma anche in presenza di input di lunghezza elevata.

Questo risultato evidenzia l'importanza di adottare pratiche di secure coding nella gestione della memoria, sottolineando come il controllo dell'input rappresenti un elemento fondamentale per la sicurezza delle applicazioni.