Build week 2

Progetto 3

Introduzione: l'obiettivo principale dell'esercizio è stato duplice:

- **Analisi del codice**: comprendere il funzionamento di un programma senza eseguirlo e identificare una potenziale vulnerabilità.
- **Sfruttamento pratico**: modificare il codice per riprodurre un errore di segmentazione, dimostrando come un'insufficiente validazione dell'input possa portare a un comportamento inatteso e critico del programma.

Analisi del codice

```c
#include <stdio.h>

int main () {

    int vector [10], i, j, k;
    int swap_var;

    printf ("Inserire 10 interi:\n");

    for ( i = 0 ; i < 10 ; i++)
    {
        int c= i+1;
        printf("[%d]:", c);
        scanf ("%d", &vector[i]);
    }

    printf ("Il vettore inserito e':\n");
    for ( i = 0 ; i < 10 ; i++)
    {
        int t= i+1;
        printf("[%d]: %d", t, vector[i]);
        printf("\n");
    }

    for (j = 0 ; j < 10 - 1; j++)
    {
        for (k = 0 ; k < 10 - j - 1; k++)
        {
            if (vector[k] > vector[k+1])
            {
                swap_var=vector[k];
                vector[k]=vector[k+1];
                vector[k+1]=swap_var;
            }
        }
    }
    printf("Il vettore ordinato e':\n");
    for (j = 0; j < 10; j++)
    {
        int g = j+1;
        printf("[%d]:", g);
        printf("%d\n", vector[j]);
    }

    return 0;
}
```

Funzionamento del codice di base

Il programma è un semplice script in C che ordina un array di numeri interi. Dichiarato un array di 10 elementi, chiede all'utente di inserire 10 numeri. Successivamente stampa l'array non ordinato e procede con un algoritmo di Bubble Sort per ordinare i numeri in ordine crescente. Infine, stampa il risultato ordinato. A livello funzionale, il programma è progettato per operare su un set fisso di 10 numeri interi e completare l'ordinamento senza errori.

Individuazione della vulnerabilità

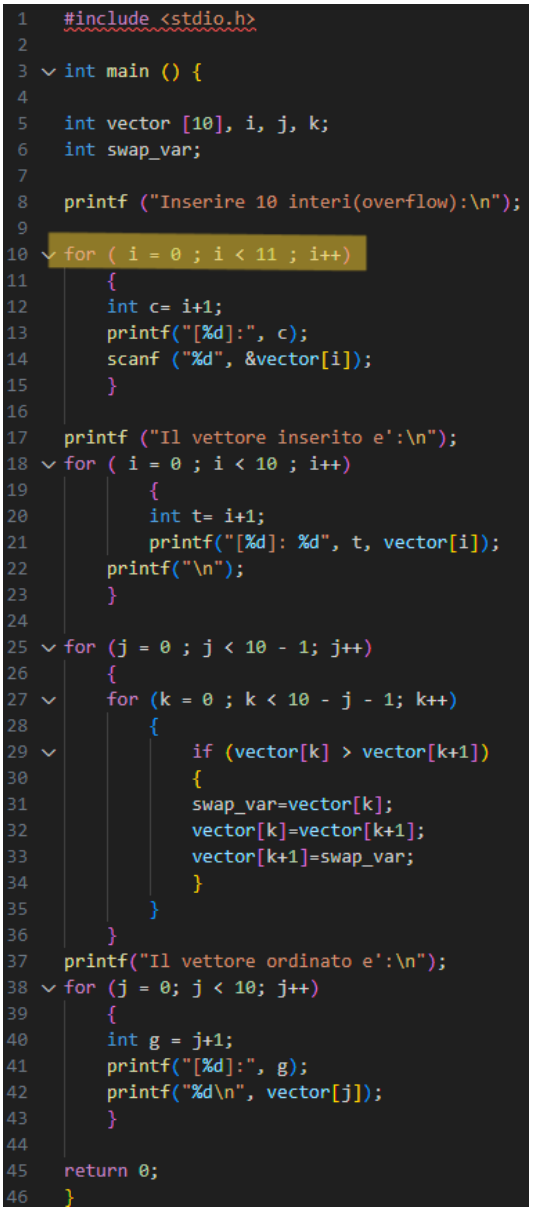
La vulnerabilità risiede nella parte del codice che gestisce l'input utente. Sebbene il ciclo for sia impostato per iterare 10 volte, la funziona scanf legge i valori direttamente nella memoria dell'array vector senza un controllo aggiuntivo sul numero di input. Se un utente (o un attaccante) modifica il programma per consentire l'inserimento di più di 10 valori, i dati in eccesso verranno scritti in posizioni di memoria oltre i confini dell'array.

Spiegazione del BOF

Questa situazione causa un Buffer Overflow (BOF). Un buffer è un'area di memoria temporanea utilizzata per memorizzare i dati. Un overflow si verifica quando un programma cerca di scrivere più dati in un buffer di quanti ne possa contenere. Questo provoca una sovrascrittura della memoria adiacente, che può corrompere i dati, o nei casi più gravi, sovrascrivere l'indirizzo di ritorno di una funzione. Quando il programma tenta di eseguire il codice all'indirizzo corrotto, si verifica un errore di segmentazione, che causa il crash del programma. Questo è il risultato finale che dimostra la vulnerabilità.

Sfruttamento della vulnerabilità

Il codice è stato modificato per trasformare il programma funzionale in un'applicazione vulnerabile al buffer overflow. La differenza principale si trova in un singolo, piccolo cambiamento nel ciclo che gestisce l'input dell'utente. Nel codice originale, il ciclo for è configurato per leggere esattamente 10 numeri interi, che è la dimensione massima del'array vector. Questo impedisce che qualsiasi scrittura oltre i limiti del buffer. Il codice modificato, invece, è stato alterato per iterare 11 volte, forzando il programma a leggere un valore in più rispetto a quanto l'array può contenere.



```c
#include <stdio.h>

int main () {

int vector [10], i, j, k;
int swap_var;

printf ("Inserire 10 interi(overflow):\n");

for ( i = 0 ; i < 11 ; i++)
    {
    int c= i+1;
    printf("[%d]:", c);
    scanf ("%d", &vector[i]);
    }

printf ("Il vettore inserito e':\n");
for ( i = 0 ; i < 10 ; i++)
        {
        int t= i+1;
        printf("[%d]: %d", t, vector[i]);
    printf("\n");
    }

for (j = 0 ; j < 10 - 1; j++)
    {
    for (k = 0 ; k < 10 - j - 1; k++)
        {
            if (vector[k] > vector[k+1])
            {
            swap_var=vector[k];
            vector[k]=vector[k+1];
            vector[k+1]=swap_var;
            }
        }
    }
printf("Il vettore ordinato e':\n");
for (j = 0; j < 10; j++)
    {
    int g = j+1;
    printf("[%d]:", g);
    printf("%d\n", vector[j]);
    }

return 0;
}
```

Questa singola modifica sfrutta la mancanza di controlli sull'input del programma. L'atto di scrivere l'undicesimo valore nell'array vector causa l'overflow dei dti che vanno a trascrivere la memoria adiacente sullo stack. Questa corruzione della memoria porta a un comportamento imprevedibile.

Compilazione ed esecuzione del codice

La fase di compilazione è stata cruciale per l'esercizio, in quanto le impostazioni predefinite del compilatore avrebbero probabilmente impedito l'errore di segmentazione. Per dimostrare in modo affidabile la vulnerabilità, il codice è stato compilato con dei flag specifici che disabilitano le protezioni e attivano i sanitizer, strumenti diagnostici che rilevano errori di memoria.

Il seguente comando è stato utilizzato nel terminale di Kali Linux per compilare il file sorgent BOF2.c in un eseguibile chiamato BOF2:

"gcc -std=c11 -Wall -Wextra -Wpedantic -O0 -g -fsanitize=address,undefined BOF2.c -o BOF2"

La scelta di questa specifica combinazione di flag è stata cruciale per la riuscita dell'esercizio. Mentre i flag come "*–Wall"* e "*–Wextra"* attivano avvisi per individuare problemi di programmazione, i flag "*–O0*" e "*–g*" disabilitano le ottimizzazioni e aggiungono informazioni di debug, rendendo più facile l'analisi del programma. Il flag più importante è stato "*–fsanitize=address,undefined*". Questo ha attivato i Sanitizer, potenti strumenti di diagnostica che monitorano l'uso della memoria in tempo reale. In particolare, l'Address Sanitizer (ASan) è stato utilizzato per intercettare l'operazione di scrittura che eccedeva i limiti dell'array. In un ambiente di produzione, tale overflow potrebbe non causare un errore visibile, ma l'ASan ha forzato il programma a terminare e a generare un report di errore dettagliato, mostrando la linea di codice esatta che ha causato il problema. In questo modo, la compilazione non ha semplicemente disattivato le sicurezze, ma ha creato un programma che era progettato per fallire in modo controllato, fornendo una prova concreta e inconfutabile che il buffer overflow è avvenuto.

Bonus

Per completare il bonus dell'esercizio, ho modificato il codice originale per aggiungere controlli sull'input e offrire all'utente la possibilità di scegliere tra una versione sicura e una vulnerabile. Il codice è stato strutturato in tre parti principali:

- La funzione bubble_sort(), che contiene il codice originale per l'ordinamento sicuro.
- La funzione overflow(), che contiene il codice vulnerabile per dimostrare il buffer overflow.
- La funzione main(), che agisce da menu per permettere all'utente di scegliere quale modalità eseguire.

Come richiesto sono stati implementati dei controlli di sicurezza sull'input per rendere il programma robusto e meno vulnerabili a malfunzionamenti causati da input non validi. L'approccio utilizzato si basa sull'uso combinato della funzione scanf() e di un ciclo while per validare ogni dato inserito dall'utente.

Nel main è stato inserito prima dello switch. Se l'input non è un numero o non rientra tra le opzioni valide, un messaggio di errore appare a schermo. In seguito, il buffer di input viene svuotato per evitare loop infiniti. Nelle funzioni bubble_sort e overflow è stato inserito un controllo simile all'interno dei cicli for per assicurarsi che vengano immessi solo numeri interi. Se l'utente digita un carattere o una stringa, il programma lo

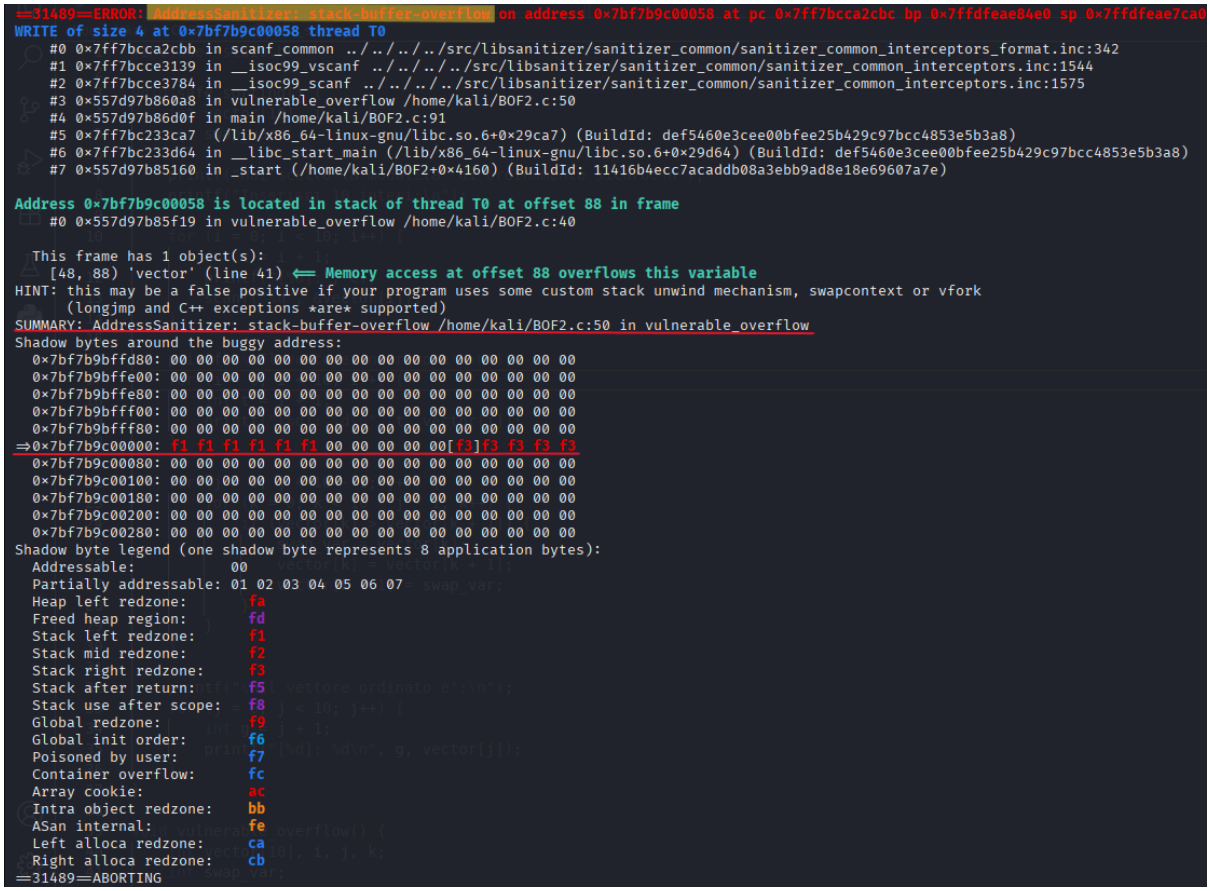
avvisa e chiede un nuovo input, prevenendo un blocco del programma. Questi controlli, però, non risolvono la vulnerabilità del buffer overflow.

```c
#include <stdio.h>

void bubble_sort() {
    int vector[10], i, j, k;
    int swap_var;

    printf("Hai scelto la modalita' di ordinamento sicuro.\n");
    printf("Inserisci 10 interi:\n");

    for (i = 0; i < 10; i++) {
        int c = i + 1;
        printf("[%d]:", c);
        while (scanf("%d", &vector[i]) != 1) {
            printf("Input non valido. Inserisci un numero intero: ");
            while (getchar() != '\n');
        }
    }

    printf("\nIl vettore inserito e':\n");
    for (i = 0; i < 10; i++) {
        int t = i + 1;
        printf("[%d]: %d\n", t, vector[i]);
    }

    for (j = 0; j < 10 - 1; j++) {
        for (k = 0; k < 10 - j - 1; k++) {
            if (vector[k] > vector[k + 1]) {
                swap_var = vector[k];
                vector[k] = vector[k + 1];
                vector[k + 1] = swap_var;
            }
        }
    }

    printf("\nIl vettore ordinato e':\n");
    for (j = 0; j < 10; j++) {
        int g = j + 1;
        printf("[%d]: %d\n", g, vector[j]);
    }
}

```

```c
void overflow() {
    int vector[10], i;

    printf("Hai scelto la modalita' vulnerabile.\n");
    printf("Inserisci 11 interi (causera' un errore di overflow):\n");

    for (i = 0; i < 11; i++) {
        int c = i + 1;
        printf("[%d]:", c);
        while (scanf("%d", &vector[i]) != 1) {
            printf("Input non valido. Inserisci un numero intero: ");
            while (getchar() != '\n');
        }
    }
}

int main() {
    int scelta;

    printf("Scegli una modalita':\n");
    printf("1. Ordinamento Sicuro\n");
    printf("2. Ordinamento Vulnerabile (Causa Overflow)\n");
    printf("Inserisci la tua scelta: ");

    while (scanf("%d", &scelta) != 1 || (scelta != 1 && scelta != 2)) {
        printf("Scelta non valida. Inserisci 1 o 2: ");
        while (getchar() != '\n');
    }

    switch (scelta) {
        case 1:
            bubble_sort();
            break;
        case 2:
            overflow();
            break;
    }

    return 0;
}
```

Esecuzione del codice

Per dimostrare l'efficacia delle modifiche apportate al codice, ho proceduto all'esecuzione del programma sul terminale di Kali. Dopo aver avviato il programma, quando mi è stato richiesto di inserire i 10 numeri interi, ne ho immesso un undicesimo. A quel punto, l'Address Sanitizer è entrato in azione. Il programma ha interrotto l'esecuzione e ha generato a schermo un report dettagliato che ha fornito una prova inequivocabile del problema.



Analisi del report di ASan

Durante l'esecuzione del programma modificato, l'Address Sanitizer ha rilevato e segnalato un errore di tipo "*stack-buffer-overflow*". L'errore è stato registrato alla riga 50 del file BOF2.c, all'interno della funzione vulnerable_overflow(). Il messaggio riportava:

"ERROR: AddressSanitizer: stack-buffer-overflow on address ..."

"SUMMARY: AddressSanitizer: stack-buffer-overflow in vulnerable_overflow /home/kali/BOF2.c:50"

ASan ha inoltre individuato il responsabile

“[48, 88) 'vector' (line 41)“

“Memory access at offset 88 overflows this variable”

Il primo messaggio di ASan specifica che la variabile *vector*, dichiarata alla riga 41 del file sorgente, occupa nello stack l’intervallo di memoria compreso tra gli offset 48 e 88 del frame della funzione. L’intervallo in questione segue la convenzione inclusivo-esclusivo (48 incluso nell’intervallo, 88 escluso dall’intervallo), quindi rappresenta 40 byte di spazio effettivamente allocato. Considerando che *vector* è un array di 10 interi che, nel sistema in uso, un int occupa 4 byte, il calcolo torna perfettamente.

La seconda riga indica che il programma ha tentato di scrivere all’indirizzo di memoria immediatamente successivo alla fine dell’array (offset 88), cioè oltre i limiti riservati a *vector*. In termini pratici, il programma ha cercato di inserire un undicesimo intero in un buffer predisposto per contenerne solo 10.

Interpretazione dei “shadow bytes”

L’Address Sanitizer utilizza una tecnica nota come shadow memory per monitorare lo stato della memoria del programma in tempo reale. In, pratica ogni intervallo di memoria reale viene “mappato” su una regione parallela detta memoria ombra, in cui vengono registrate informazioni sullo stato di validità dei byte corrispondenti. Grazie a questo meccanismo, ASan è in grado di intercettare immediatamente accessi non validi, come letture o scritture oltre i limiti di un buffer. Nel report viene mostrato un dump esadecimale della memoria, con valori speciali che rappresentano diversi stati della memoria:

=>0x7bf7b9c00000: f1 f1 f1 f1 f1 00 00 00 00 00 [f3] f3 f3 f3 f3

- I primi 5 byte f1 corrispondono alla *stack left redzone*, un’area sentinella che ASan inserisce immediatamente prima della variabile per intercettare eventuali scritture in *underflow*. Questa zona non deve mai essere toccata dal programma.
- I successivi 5 byte 00 rappresentano la memoria assegnata alla variabile *vector*. Questi byte sono gli unici legittimamente accessibili dal programma.
- Il simbolo [f3] indica il punto preciso in cui il programma ha effettuato un accesso non valido.
- I rimanenti 4 byte f3 costituiscono la *stack right redzone*, un’altra area sentinella collocata subito dopo la variabile per rilevare scritture in *overflow*.

La riga del dump, in sintesi, dimostra con chiarezza che il programma ha provato a scrivere un undicesimo elemento oltre i 10 byte previsti.