



Build Week 2 – Progetto 3

Sistemi Exploit BOF

OBIETTIVO DELL'ATTIVITÀ

analizzare il funzionamento del codice e sperimentare come l'assenza di controlli sui limiti dell'array possa portare a errori di segmentazione e vulnerabilità di tipo Buffer Overflow.

EXECUTIVE SUMMARY

1. Descrizione del funzionamento (Analisi Statica)

In questa fase effettueremo un'analisi teorica del codice sorgente. L'obiettivo è mappare l'intero flusso logico del programma

2. Esecuzione del programma (Analisi Dinamica)

Passeremo alla fase operativa compilando il sorgente con Visual Studio Code ed eseguendolo in un ambiente controllato. Verificheremo che il comportamento del software corrisponda a quanto analizzato in precedenza.

3. Manipolazione del codice e Segmentation Fault

Introdurremo intenzionalmente una vulnerabilità di tipo Buffer Overflow modificando i limiti del ciclo di scrittura. Estendendo l'input oltre la capacità dell'array (che è di 10 elementi) e osserveremo la corruzione della memoria.

4. Conclusioni

Il capitolo finale riassume i risultati ottenuti, evidenziando la pericolosità delle funzioni di input non protette in C (come scanf) e come la mancanza di controllo dei limiti (bounds checking) sia la causa principale delle vulnerabilità che permettono l'esecuzione di codice arbitrario o il crash del sistema.

1. Descrizione del funzionamento del programma prima dell'esecuzione.

Obiettivo: analizzare il codice sorgente del programma senza eseguirlo.

Questo codice è scritto in linguaggio **C** ed esegue l'ordinamento di un array di numeri interi inseriti dall'utente. Nello specifico, il programma segue questi passaggi:

1. **Input:** richiede all'utente di inserire **10 numeri interi**;
2. **Visualizzazione:** stampa il vettore così come è stato inserito;
3. **Ordinamento:** usa un algoritmo per ordinare i numeri in ordine crescente;
4. **Output:** stampa il vettore finale ordinato.

L'algoritmo funziona confrontando ripetutamente coppie di elementi adiacenti e scambiandoli se sono nell'ordine errato. Se l'utente inserisse caratteri non numerici o input inaspettati, il programma potrebbe entrare in un loop infinito o comportarsi in modo instabile. Non sembra esserci un rischio diretto di Buffer Overflow, dato che il ciclo for è limitato rigidamente a 10 iterazioni.

2. Esecuzione del programma nel laboratorio

Obiettivo: verificare la correttezza delle ipotesi in fase di analisi del programma prima dell'esecuzione.

In questa fase, il codice sorgente *BW_D3_BOF.c* è stato sottoposto a test dinamico per verificarne la correttezza logica in un ambiente operativo.

Procedura di test:

1. **Compilazione:** il programma è stato compilato utilizzando il compilatore *gcc* tramite la riga di comando. Non sono stati riscontrati errori di sintassi o warning.
2. **Input fornito:** durante l'esecuzione, sono stati inseriti 10 numeri interi in ordine casuale come richiesto dal prompt del terminale.
3. **Verifica del flusso:** il software ha correttamente memorizzato i dati nel vettore, li ha stampati a video per conferma e ha avviato la routine di ordinamento.

```
● └─(kali㉿kali)-[~/Desktop/c-programs]
$ ./bw2test
Inserire 10 interi:
[1]:3
[2]:5
[3]:6
[4]:10
[5]:12
[6]:15
[7]:4
[8]:2
[9]:1
[10]:20
Il vettore inserito e':
[1]: 3
[2]: 5
[3]: 6
[4]: 10
[5]: 12
[6]: 15
[7]: 4
[8]: 2
[9]: 1
[10]: 20
Il vettore ordinato e':
[1]:1
[2]:2
[3]:3
[4]:4
[5]:5
[6]:6
[7]:10
[8]:12
[9]:15
[10]:20
```

Risultati osservati: Come mostrato nello screenshot, il programma ha restituito il vettore perfettamente ordinato in modo crescente. L'algoritmo ha operato esattamente come previsto durante l'analisi teorica preliminare, effettuando gli scambi necessari tra gli elementi adiacenti senza produrre anomalie.

Valutazione della stabilità: In questa condizione di utilizzo standard (rispettando il limite di 10 input), il programma si è dimostrato stabile. L'area di memoria allocata per

l'array vector[10] è risultata sufficiente a contenere i dati, evitando interferenze con il resto dello stack di memoria del processo.

3. Modificare il programma affinché si verifichi un errore di segmentazione.

Obiettivo: forzare un **Segmentation Fault**, analizzando come il sistema operativo interrompa il processo quando si tenta di accedere a indirizzi di memoria non autorizzati.

Cos'è un Segmentation Fault: è un errore di segmentazione (abbreviato in *segfault*) è un segnale inviato dal sistema operativo a un programma che tenta di accedere a una zona della memoria RAM che non gli appartiene o a cui non ha i permessi per accedere.

In parole povere: il programma ha cercato di "scavalcare il recinto" della sua memoria riservata.

Quando si verifica: in C, si verifica principalmente in tre scenari:

- **Buffer Overflow:** quando si scrivono dati oltre il limite di un array, andando a colpire indirizzi di memoria riservati ad altre variabili o al sistema. Come in questo caso
- **Puntatori Nulli o Non Inizializzati:** quando si cerca di leggere o scrivere un indirizzo di memoria NULL o un indirizzo casuale contenuto in un puntatore non impostato.
- **Accesso a memoria protetta:** quando il programma tenta di scrivere in una zona di sola lettura (come il segmento di codice dove risiedono le istruzioni).

Modifica

L'errore critico si trova all'interno della funzione **vulnerabile()**, specificamente in questo ciclo **for**:

```
// Buffer Overflow

printf("Inserisci i numeri:\n");
for (i = 0; i < numb; i++) {
    printf("Numero [%d]: ", i + 1);
    scanf("%d", &vector[i]);
}
```

Il punto esatto del problema è l'istruzione `scanf("%d", &vector[i]);` quando la variabile `i` raggiunge il valore **10**.

Dettagli della Vulnerabilità:

1. Viene allocato nello stack un array statico, `vector[10]`, con una dimensione fissa di 10 interi.
2. L'utente può definire arbitrariamente il numero di scritture tramite la variabile `numb`.
3. Il linguaggio **C** non esegue controlli automatici sui limiti degli array (**bounds checking**).

Conseguenze dell'Overflow:

Se l'utente specifica un valore `numb > 10`, il ciclo `for` continua a scrivere dati al di fuori dei limiti consentiti del vettore. Questa scrittura fuori dai limiti (**overflow**) sovrascrive dati essenziali nello stack. La **corruzione dello stato del programma** che ne deriva conduce a un **Segmentation Fault** o, in contesti di attacco, alla potenziale **esecuzione di codice arbitrario**.

Inserimento controlli di INPUT

Come richiesto dalla traccia bonus, è stato implementato un controllo di input che verifica quanti numeri vengono inseriti.

```
-MODALITA' SICURA-
Il programma puo' gestire un massimo di 10 numeri.
Quanti numeri vuoi inserire? 20
ERRORE: Non puoi inserire più di 10 numeri!
```

Possiamo vedere come il programma restituisce errore dopo aver indicato l'inserimento di 20 numeri.

Implementazione di un menù di scelta tra il programma che va in errore e quello corretto

```
-BUFFER OVERFLOW-
Scegli cosa fare:
1. Esegui versione SICURA (con controlli)
2. Esegui versione VULNERABILE
Scelta: 2

-MODALITA' VULNERABILE-
Il programma ha spazio per 10 numeri.
Quanti numeri vuoi inserire? (Scrivi un numero maggiore di 10 per causare l'errore): 50
Inserisci i numeri:
Numero [1]: 4
Numero [2]: 5
Numero [3]: 3
Numero [4]: 6
Numero [5]: 3
Numero [6]: 67
Numero [7]: 2
Numero [8]: 77
Numero [9]: 34
Numero [10]: 235
Numero [11]: 5
Numero [12]: 323
Numero [13]: 535
Numero [14]: 34432
Numero [15]: 2334
Numero [16]: 25443
234

4324
```

4. Conclusioni

L'attività di laboratorio ha permesso di analizzare concretamente come la gestione della memoria in linguaggio C possa diventare un vettore di attacco se non correttamente presidiata.

Punti chiave emersi:

- **Assenza di Bounds Checking:** è stato dimostrato che il compilatore C non impedisce la scrittura di dati oltre i limiti allocati per un array. La responsabilità della sicurezza della memoria ricade interamente sullo sviluppatore;
- **Meccanismi di Crash:** il *Segmentation Fault* osservato non è solo un errore software, ma un meccanismo di protezione del Sistema Operativo che interviene per prevenire danni maggiori quando un processo tenta di corrompere segmenti di memoria non autorizzati;
- **L'importanza della Validazione:** l'implementazione del controllo sull'input e del menu di scelta ha evidenziato la differenza tra un codice puramente funzionale e un codice "sicuro". Limitare il numero di inserimenti alla dimensione reale del buffer è la difesa primaria contro gli attacchi di tipo *Stack Smashing*.