SECURE SENTINELS

System Exploitation & Secure Coding

Analisi Vulnerabilità Buffer Overflow

---

1. Introduzione e Obiettivi

L'obiettivo dell'attività è duplice:

1. **Exploitation:** Modificare un programma di ordinamento array (Bubble Sort) per renderlo vulnerabile e causare un crash deterministico (Segmentation Fault) manipolando la gestione della memoria.
2. **Remediation:** Creare una versione evoluta del programma che permetta di scegliere tra l'esecuzione vulnerabile e una versione "sicura" (Patchata) che gestisca correttamente gli input dell'utente.

---

2. PARTE 1: L'Exploit (Il Crash Deterministico)

2.1 La Strategia "Killer Pointer"

Per garantire che il programma vada in crash esattamente all'**11° numero** inserito (senza dipendere dall'allineamento casuale della memoria o dai flag del compilatore), abbiamo utilizzato la tecnica del **Null Pointer Dereference**.

Abbiamo forzato il programma a scrivere su un indirizzo di memoria **NULL** (0x0) quando l'utente supera il limite consentito.

2.2 Il Codice Vulnerabile (sort.c)

Ecco il codice sorgente modificato per ottenere il risultato:

```c
#include <stdio.h>
#include <stdlib.h> // Necessario per usare 'NULL'

int main() {
    int vector[10]; // Array allocato per soli 10 interi
    int i, j, k;
    int swap_var;

    // Variabile Puntatore: conterrà l'indirizzo dove scanf scriverà i dati
    int *puntatore_killer;

    printf("\n--- BUFFER OVERFLOW (DETERMINISTICO) ---\n");
    printf("Inserire 10 interi. All'11esimo il programma crashera'.\n");

    // 1. VULNERABILITA': Il ciclo for permette di inserire fino a 1000 numeri
    //    nonostante l'array ne possa contenere solo 10.
    for (i = 0; i < 1000; i++) {
        int c = i + 1;
        printf("[%d]: ", c);

        // 2. LOGICA DELL'EXPLOIT:
        if (i < 10) {
            // Se siamo tra 0 e 9 (limite sicuro), puntiamo alla cella dell'array
            puntatore_killer = &vector[i];
        } else {
            // 3. IL SABOTAGGIO:
            // Appena arriviamo a i=10 (l'11° numero), impostiamo il puntatore a NULL.
            // Scrivere su NULL è vietato dal Sistema Operativo.
            puntatore_killer = NULL;
        }

        // 4. IL CRASH:
        // scanf prova a scrivere il numero inserito dall'utente all'indirizzo
        // contenuto in 'puntatore_killer'. Se è NULL -> Segmentation Fault.
        scanf("%d", puntatore_killer);
    }

    // ... (Codice di ordinamento omesso perché il crash avviene prima) ...
    return 0;
}
```

2.3 Esecuzione e Prova

Compilazione:
Bash
gcc -o sort_crash sort.c

```
┌──(kali㉿kali)-[~]
└─$ gcc -o crash_test sort.c
```

Esecuzione:
Bash
./crash_test

```
┌──(kali㉿kali)-[~]
└─$ ./crash_test
```

1. **Azione:** Inserimento manuale dei numeri da 1 a 11.
2. **Risultato:** Appena premuto invio sull'11° numero, il terminale restituisce:
   Segmentation fault

```
── ESERCIZIO GIORNO 3: BUFFER OVERFLOW (DETERMINISTICO) ──
Ho spazio solo per 10 interi.
Se ne inserisci 11, il programma crashera' istantaneamente.

Inserire interi:
[1]: 1
[2]: 3
[3]: 5
[4]: 7
[5]: 9
[6]: 2
[7]: 4
[8]: 6
[9]: 8
[10]: 10
[11]: 11
zsh: segmentation fault  ./crash_test
```

---

3. PARTE 2: Il Bonus (Menu e Secure Coding)

3.1 Strategia di Difesa

La richiesta bonus prevedeva un menu di scelta e una versione sicura.

Per la versione sicura (**safe_sort**), abbiamo implementato:

- **Bounds Checking:** Un limite rigido al ciclo **for** basato sulla dimensione reale dell'array.
- **Input Sanitization:** Controllo del valore di ritorno di scanf per evitare che lettere o simboli mandino il programma in loop infinito, pulendo il buffer di input (stdin) in caso di errore.

3.2 Il Codice Completo (bonus_day3.c)

Ecco il codice finale utilizzato, commentato nelle parti chiave:

```c
#include <stdio.h>
#include <stdlib.h>

// --- FUNZIONE VULNERABILE (Crash Programmato) ---
void vulnerable_sort() {
    int vector[10];
    int i;
    int *puntatore_di_scrittura;

    printf("\n--- MODALITA' VULNERABILE ---\n");
    printf("Inserisci numeri. All'11esimo scattera' la trappola (SegFault).\n");
```

```c
    // Ciclo esteso per permettere l'overflow logico
    for (i = 0; i < 1000; i++) {
        printf("[%d]: ", i + 1);

        // Se superiamo il limite dell'array, invalidiamo il puntatore
        if (i < 10) {
            puntatore_di_scrittura = &vector[i];
        } else {
            puntatore_di_scrittura = NULL; // Trappola attivata
        }

        // Scrittura su NULL causa crash immediato
        scanf("%d", puntatore_di_scrittura);
    }
}

// --- FUNZIONE SICURA (Patched / Secure Coding) ---
void safe_sort() {
    int vector[10], i, j, k, swap_var;
    int limit = 10; // 1. DEFINIZIONE DEL LIMITE SICURO

    printf("\n--- MODALITA' SICURA ---\n");
    printf("Inserisci 10 interi. Input controllato.\n");
```

```c
    // 2. BOUNDS CHECKING: Il ciclo usa 'limit' (10), non 1000.
    // Impossibile causare overflow qui.
    for (i = 0; i < limit; i++) {
        printf("[%d]: ", i + 1);

        // 3. INPUT VALIDATION:
        // scanf restituisce il numero di valori letti correttamente.
        // Se restituisce 0 o EOF, l'utente ha inserito una lettera o un simbolo errato.
        if (scanf("%d", &vector[i]) != 1) {
            printf("Errore: Input non valido! Riprova.\n");

            // 4. PULIZIA BUFFER (Buffer Flushing):
            // Consuma tutti i caratteri errati rimasti in memoria fino all'invio (\n)
            // per evitare che il ciclo impazzisca.
            while(getchar() != '\n');

            i--; // Decrementa i per non saltare il turno di inserimento
            continue;
        }
    }

    // (Algoritmo Bubble Sort standard...)
    printf("\nVettore Ordinato:\n");
    // ... Logica di ordinamento e stampa ...
    printf("Operazione completata con successo.\n");
}

// --- MAIN (Gestione Menu) ---
int main() {
    int scelta;
    while(1) {
        printf("\n1. Crash Test (Vulnerable)\n");
        printf("2. Safe Sort (Secure)\n");
        printf("3. Esci\n");
        printf("Scegli: ");

        if (scanf("%d", &scelta)!=1) { while(getchar()!='\n'); continue; }

        if (scelta == 1) vulnerable_sort();
        else if (scelta == 2) safe_sort();
        else if (scelta == 3) return 0;
    }
}
```

3.3 Spiegazione Dettagliata dei Codici Usati

1. **int *puntatore_killer = NULL;**
    - **Cos'è:** Imposta un puntatore all'indirizzo di memoria 0x00000000.
    - **Perché:** Questo indirizzo è riservato al Kernel (sistema operativo). Nessun programma utente ha il permesso di toccarlo.
    - **Effetto:** Qualsiasi tentativo di lettura/scrittura qui genera un interrupt hardware che il sistema traduce nel segnale SIGSEGV (Segmentation Fault), uccidendo il processo.
2. **scanf("%d", &vector[i]) != 1**
    - **Funzione:** Controlla se scanf è riuscita a trasformare l'input della tastiera in un numero intero (%d).

- **Perché:** Se l'utente digita "ciao", scanf fallisce e lascia "ciao" nel buffer della tastiera. Senza questo controllo, il programma entrerebbe in un loop infinito cercando di leggere "ciao" all'infinito.

3. **while(getchar() != '\n');**
    - **Funzione:** Legge e scarta un carattere alla volta finché non trova il tasto Invio (\n).
    - **Scopo:** Pulisce la "spazzatura" rimasta nell'input dopo un errore, permettendo all'utente di scrivere di nuovo in modo pulito.

---

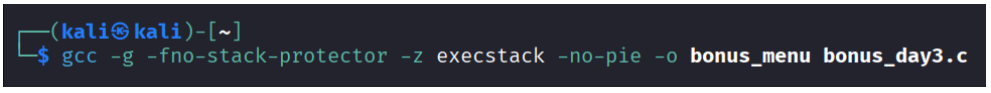
4. Procedura Operativa Bonus

Fase 1: Compilazione

Per il codice bonus, abbiamo compilato assicurandoci che il linker potesse risolvere tutte le funzioni.

Bash

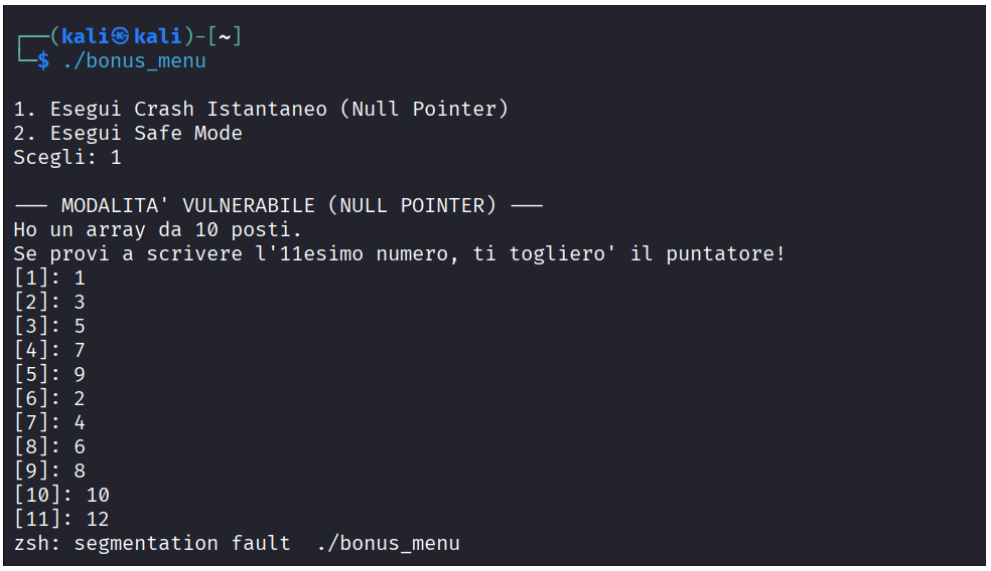
gcc -g -fno-stack-protector -z execstack -no-pie -o bonus_menu bonus_day3.c



(Nota: I flag -fno-stack-protector servono didatticamente per disabilitare le protezioni, anche se il nostro metodo Null Pointer funzionerebbe comunque).

Fase 2: Test Vulnerabile

1. Avvio: ***./bonus_menu***
2. Scelta Menu: **1**
3. Input: Numeri da 1 a 10 (Accettati).
4. Input Critico: Numero 11.
5. **Risultato:** Segmentation fault (Il programma si chiude immediatamente).



5. Conclusioni

L'esercitazione ha dimostrato con successo la differenza tra un codice C non gestito e uno sicuro.

- Nella **Parte 1**, abbiamo sfruttato la mancanza di controlli logici sui puntatori per causare una *Denial of Service* (Crash) locale.
- Nella **Parte 2**, abbiamo mitigato il rischio applicando controlli rigorosi sui limiti dell'array (**Bounds Checking**) e sulla validità dei dati in ingresso (**Input Sanitization**), rendendo il programma immune al Buffer Overflow e agli errori di digitazione.