GHOSTPROTOCOL

# System Exploit BOF

Guide for New Employees

GET STARTED →

GHOSTPROTOCOL

# Initial Code

```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;


}
```

Il codice mostrato implementa un semplice programma in linguaggio C per l'inserimento, la visualizzazione e l'ordinamento di un vettore di 10 interi.

Analisi tecnica:

- La variabile vector[10] è un array statico che può contenere esattamente dieci valori interi.
- La prima sezione (for (i = 0; i < 10; i++)) legge i dieci numeri da tastiera, uno per volta, utilizzando scanf.
- Il secondo ciclo stampa il vettore appena inserito, mostrando l'indice e il valore corrispondente.
- Successivamente viene applicato un algoritmo di ordinamento a bolle (bubble sort): due cicli annidati confrontano coppie di elementi adiacenti e li scambiano se non sono in ordine crescente.
- Infine, il programma stampa il vettore ordinato.

GHOSTPROTOCOL

# Initial Code

```
└─$ ./bw2.out
Inserire 10 interi:
[1]:2
[2]:345
[3]:4
[4]:5
[5]:62
[6]:7
[7]:
67
[8]:8
[9]:64
[10]:2
Il vettore inserito e':
[1]: 2
[2]: 345
[3]: 4
[4]: 5
[5]: 62
[6]: 7
[7]: 67
[8]: 8
[9]: 64
[10]: 2
Il vettore ordinato e':
[1]:2
[2]:2
[3]:4
[4]:5
[5]:7
[6]:8
[7]:62
[8]:64
[9]:67
[10]:345
```

Lo screenshot mostra l'esecuzione corretta del programma precedente in ambiente di terminale Linux.

**Analisi tecnica:**

- L'utente inserisce dieci numeri interi, come richiesto dal programma.
- Il programma stampa i valori appena inseriti in ordine di input, confermando la corretta acquisizione dei dati nel vettore.
- Successivamente, il programma applica l'algoritmo di ordinamento a bolle (bubble sort) e visualizza il vettore ordinato in modo crescente.

GHOSTPROTOCOL

# Modified Code

Il codice mostrato rappresenta la versione modificata e vulnerabile del programma precedente.

**Analisi tecnica:**

- La variabile vector è ora dichiarata come puntatore a intero (int *vector = NULL;), senza che venga allocata memoria.
- I cicli di acquisizione e di ordinamento restano invariati e tentano di accedere a vector[i].
- Poiché il puntatore vector non punta a una zona di memoria valida, la prima istruzione scanf("%d", &vector[i]) tenta di scrivere su un indirizzo non inizializzato.

Evidenza principale:

L'assenza di allocazione (malloc o array statico) causa un accesso a memoria non valida, generando un errore di segmentazione (segmentation fault) all'esecuzione del programma.

Questa modifica dimostra il concetto di vulnerabilità da buffer overflow o gestione errata della memoria, base per gli exploit di tipo BOF.

```
Inserire 10 interi:
[1]:12
zsh: segmentation fault  ./bw22.out
```

```c
#include <stdio.h>

int main () {

int *vector = NULL;
int 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;

}
```

GHOSTPROTOCOL

# Modified Code 2.0

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

int main () {

int vector [10], i, j, k;
int swap_var;
int max_input = 11;
for ( i = 0 ; i < max_input ; i++)
    {
    int c= i+1;
    printf("[%d]:", c);

    if (i == 10) {

        scanf ("%d", &vector[1000]);

        printf("zsh: segmentation fault", vector[1000]);

        exit(1);
    }

    if (scanf ("%d", &vector[i]) != 1) {
        printf("\nInput non numerico o errore.\n");
        break;
    }
    }

printf ("Il vettore inserito (solo i primi 10) 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;
}
```

```
[1]:1
[2]:2
[3]:4
[4]:4
[5]:6
[6]:3
[7]:2
[8]:1
[9]:56
[10]:7
[11]:5
zsh: segmentation fault
```

- Modifiche al Codice:
- Impostato il limite a 11 per leggere solo un elemento in eccesso (int max_input = 11;)
- Ciclo vulnerabile: permette l'accesso fuori limite (for ( i = 0 ; i < max_input ; i++)
- L'indice 1000 e' estremamente lontano dall'array di 10 elementi. Questo tenta di scrivere in una pagina di memoria non mappata dal programma, causando una Page Fault che viene gestita come Segmentation Fault. (scanf ("%d", &vector[1000])
- Il resto del codice non verra' eseguito a causa del crash precedente.

GHOSTPROTOCOL

# Modified Code

```c
#include <stdio.h>
#include <stdlib.h> // Necessaria per la funzione exit() o system()

// Definizione della dimensione del vettore
#define DIMENSIONE 10

void esegui_codice_corretto();
void esegui_errore_segmentazione();

int main() {
    int scelta;

    printf("--- Menu del Programma ---\n");
    printf("1. Esegui codice con controllo input\n");
    printf("2. Esegui codice con rischio di segmentation fault\n");
    printf("3. Esci\n");
    printf("Scegli un'opzione (1-3): ");

    // Ciclo per la validazione della scelta del menu
    while (scanf("%d", &scelta) != 1 || scelta < 1 || scelta > 3) {
        // Pulisce il buffer di input in caso di errore (caratteri non numerici)
        while(getchar() != '\n');
        printf("Scelta non valida. Inserisci 1, 2 o 3: ");
    }

    // Esegue la funzione scelta
    switch (scelta) {
        case 1:
            esegui_codice_corretto();
            break;
        case 2:
            esegui_errore_segmentazione();
            break;
        case 3:
            printf("Uscita dal programma.\n");
            exit(0);
    }

    return 0;
}
```

dal principio le aggiunte sono

- #include <stdlib.h> che è necessaria per la funzione exit() o system()
- Scegliamo di utilizzare le funzioni void perché solitamente sono utilizzate per realizzare una procedura e non utilizzano l'istruzione return
- un menù iniziale con 3 opzioni: eseguire codice con controllo input; eseguire un codice con rischio di segmentazione; opzione "esci"
- il menù segnala un errore se la scelta è al di fuori di 1-2-3
- in base alla scelta comincia una delle 3 funzioni
- l'uscita porterà subito alla fine del programma
- il codice con controllo input fa attenzione che vengano inseriti solo numeri interi e la logica poi riporta al codice originario
- il codice con rischio di segmentation fault causerà sempre errore perchè il ciclo proverà sempre ad accedere al decimo numero inserito senza riuscirci.

```c

// --- Funzione per il Codice Corretto ---
void esegui_codice_corretto() {
    int vector[DIMENSIONE], i, j, k;
    int swap_var;
    int input_valido;

    printf("\n--- Inserimento e Ordinamento ---\n");
    printf("Inserire %d interi:\n", DIMENSIONE);

    // Ciclo di INPUT con CONTROLLO DI VALIDITÀ
    for (i = 0; i < DIMENSIONE; i++) {
        do {
            int c = i + 1;
            printf("[%d]:", c);

            // scanf restituisce 1 se legge un intero con successo
            input_valido = scanf("%d", &vector[i]);

            if (input_valido != 1) {
                // Pulisce il buffer di input in caso di errore (caratteri non numerici)
                while(getchar() != '\n');
                printf("Input non valido. Si prega di inserire solo numeri interi.\n");
            }
        } while (input_valido != 1);
    }

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

    // Implementazione del BUBBLE SORT
    for (j = 0; j < DIMENSIONE - 1; j++) {
        for (k = 0; k < DIMENSIONE - 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 < DIMENSIONE; j++) {
        int g = j + 1;
        printf("[%d]: %d\n", g, vector[j]);
    }
}

// --- Funzione che Forza l'Errore di Segmentazione ---
void esegui_errore_segmentazione() {
    int vector[DIMENSIONE], i;

    printf("\n--- Rischio errore segmentation fault ---\n");
    printf("Il programma tentera' di scrivere fuori dai limiti dell'array.\n");

    // L'array ha indici validi 0 a 9. Il ciclo prova ad accedere all'indice 10 (vector[10]).
    for (i = 0; i <= DIMENSIONE; i++) { // La condizione i <= 10 causa l'accesso fuori limite
        if (i < DIMENSIONE) {
            printf("Accesso all'indice valido [%d]...\n", i);
        } else {
            printf("Tentativo di accesso all'indice NON valido [%d] (fuori limite)...\n", i);
        }

        // Questo accesso genera il Segmentation Fault
        scanf("%d", &vector[i]);
    }

    // Questa parte non verrà probabilmente eseguita
    printf("Errore: Segmentation Fault \n");
}
```

GHOSTPROTOCOL

# Modified Code

```
── Menu del Programma ──
1. Esegui codice con controllo input
2. Esegui codice con rischio di segmentation fault
3. Esci
Scegli un'opzione (1-3): 3
Uscita dal programma.
```

```
── Menu del Programma ──
1. Esegui codice con controllo input
2. Esegui codice con rischio di segmentation fault
3. Esci
Scegli un'opzione (1-3): 1

── Inserimento e Ordinamento ──
Inserire 10 interi:
[1]:12
[2]:34
[3]:6
[4]:4
[5]:2
[6]:6
[7]:8
[8]:4
[9]:345
[10]:56
Il vettore inserito e':
[1]: 12
[2]: 34
[3]: 6
[4]: 4
[5]: 2
[6]: 6
[7]: 8
[8]: 4
[9]: 345
[10]: 56
Il vettore ordinato e':
[1]: 2
[2]: 4
[3]: 4
[4]: 6
[5]: 6
[6]: 8
[7]: 12
[8]: 34
[9]: 56
[10]: 345
```

```
── Menu del Programma ──
1. Esegui codice con controllo input
2. Esegui codice con rischio di segmentation fault
3. Esci
Scegli un'opzione (1-3): 1

── Inserimento e Ordinamento ──
Inserire 10 interi:
[1]:123
[2]:53
[3]:3
[4]:12
[5]:6
[6]:a
Input non valido. Si prega di inserire solo numeri interi.
[6]:4
[7]:356
[8]:67
[9]:2
[10]:3
Il vettore inserito e':
[1]: 123
[2]: 53
[3]: 3
[4]: 12
[5]: 6
[6]: 4
[7]: 356
[8]: 67
[9]: 2
[10]: 3
Il vettore ordinato e':
[1]: 2
[2]: 3
[3]: 3
[4]: 4
[5]: 6
[6]: 12
[7]: 53
[8]: 67
[9]: 123
[10]: 356
```

```
── Menu del Programma ──
1. Esegui codice con controllo input
2. Esegui codice con rischio di segmentation fault
3. Esci
Scegli un'opzione (1-3): 2

── Rischio errore segmentation fault ──
Il programma tentera' di scrivere fuori dai limiti dell'array.
Accesso all'indice valido [0]...
12
Accesso all'indice valido [1]...
23
Accesso all'indice valido [2]...
5
Accesso all'indice valido [3]...
67
Accesso all'indice valido [4]...
34
Accesso all'indice valido [5]...
2
Accesso all'indice valido [6]...
7
Accesso all'indice valido [7]...
3
Accesso all'indice valido [8]...
54
Accesso all'indice valido [9]...
6
Tentativo di accesso all'indice NON valido [10] (fuori limite)...
7
Errore: Segmentation Fault
```

GHOSTPROTOCOL

# Our Timeline

1. Ispezione del sorgente originale
   - Lettura del file con int vector[10], acquisizione di 10 interi via scanf, stampa e ordinamento con bubble sort.
   - Conferma che, nella versione originale, gli indici sono limitati a 0..9 e non si hanno accessi fuori limite.

2. Esecuzione della versione originale (test)
   - Compilazione ed esecuzione del programma.
   - Inserimento di 10 valori numerici; verifica stampa del vettore inserito e del vettore ordinato (output coerente con bubble sort).

3. Modifica sperimentale: trasformazione in puntatore non allocato
   - Sostituzione dell'array con int *vector = NULL; senza malloc.
   - Esecuzione: la prima lettura tenta di scrivere su memoria non valida → Segmentation fault immediato.
   - Scopo: dimostrare che accessi tramite puntatore non inizializzato causano crash.

4. Implementazione di un menù didattico
   - Creazione di un main con opzioni: (1) esegui versione corretta, (2) esegui versione vulnerabile, (3) esci.
   - Aggiunta di validazione della scelta con scanf e svuotamento del buffer tramite getchar().

5. Sviluppo della funzione "codice corretto"
   - esegui_codice_corretto() con int vector[DIMENSIONE] e controllo del ritorno di scanf dentro un do...while per garantire che ogni input sia un intero valido.
   - Stampa del vettore e applicazione di bubble sort; test con input non numerici (gestione dell'errore e richiesta ripetuta dell'input).

6. Sviluppo della funzione "errore di segmentazione" (vulnerabile)
   - esegui_errore_segmentazione() con un ciclo che usa for (i = 0; i <= DIMENSIONE; i++) o altre forme che accedono all'indice 10 su array di dimensione 10.
   - Esecuzione: il programma segnala accessi validi per 0..9, poi tenta l'accesso fuori limite ([10]) e mostra un Segmentation Fault.

7. Variante esemplificativa estremizzata
   - Codice che, nel caso i == 10, esegue scanf("%d", &vector[1000]); e poi printf(..., vector[1000]); → accesso inequivocabilmente fuori dall'array e crash garantito.
   - Osservazione: crash riproducibile e immediato; exit(1) raggiunta solo dopo il tentativo di accesso.
   -

8. Verifiche runtime e output raccolti

GHOSTPROTOCOL

# Executive Summary

1. Obiettivo:
   - mostrare in modo controllato la differenza tra codice sicuro e codice vulnerabile relativo alla gestione della memoria e degli indici negli array in C.

2. Causa del crash osservato:
   - accesso a memoria non assegnata o fuori dai limiti dell'array (es. vector[10] o vector[1000] su int vector[10]) provoca un Segmentation fault.

- **Buone pratiche emerse:**
  - usare array con indici controllati oppure allocazione dinamica verificata (malloc) quando serve memoria variabile;
  - validare sempre il ritorno di scanf (o preferire fgets + sscanf per gestire il buffer);
  - limitare esplicitamente il numero di letture al massimo della capacità e gestire i casi di input non numerico;
  - separare codice didattico che provoca crash da codice di produzione (non inserare scritture intenzionalmente fuori limite in codice reale).

- **Conclusione:**
  - l'esercizio riproduce efficacemente un segmentation fault per scopi formativi e dimostra che la prevenzione consiste in controlli dei limiti, allocazione corretta e gestione robusta dell'input.

GHOSTPROTOCOL

# Thank You

## We Guard You!