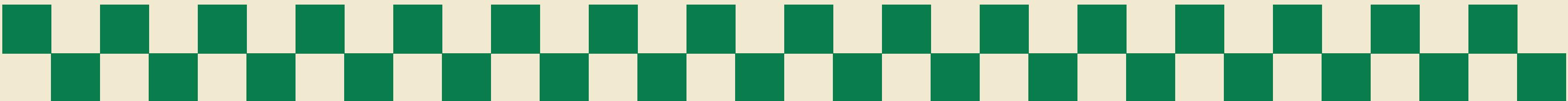


Consegna S7/L4

di Giuseppe Lupoi





Cos'è il Buffer Overflow?



Nella lezione affrontata stamane abbiamo visto più da vicino il Buffer Overflow, si tratta di una vulnerabilità che colpisce i limiti del buffer dove appunto un utente andrà ad inserire un input, questo accade per via di una mancanza di controlli proprio nel campo del buffer.



Nella traccia di oggi ci viene chiesto di prendere il codice della slide e modificarlo per evitare che si presentino errori di Buffer Overflow





Diamo un occhiata!

Questo è il codice originale riportato nella slide di EPICODE.

Utilizziamolo per capire come funziona.

Avviamo Kali, apriamo un nuovo terminale sul Desktop ed accediamo all'editor con il comando “**sudo nano**”.

Una volta trascritto il codice premiamo “**ctrl x**” per uscire, subito dopo l'editor ci chiederà se vogliamo salvare il file, clicchiamo “**y**” e nominiamolo “**BOF.c**”

The screenshot shows a Kali Linux desktop environment. In the top right corner, there's a terminal window titled "kali@kali: ~/Desktop". Below it, a nano editor window is open with the file "BOF.c" containing the following C code:

```
GNU nano 7.2          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;
}
```

At the bottom of the screen, a file dialog box is displayed, asking "File Name to Write: BOF.c". It includes standard nano key bindings: **^G Help**, **^D DOS Format**, **^A Append**, **^B Backup File**, **^C Cancel**, **M-M Mac Format**, **M-P Prepend**, and **^T Browse**.

Quindi dopo aver trascritto il codice nel nostro editor avviamo la compilazione con questo * comando “**gcc -g nome_file -o file_compilato**”.

N.B. Questa fase è fondamentale per permettere al compilatore di leggere il file, nel linguaggio C questo passaggio è necessario a differenza di altri linguaggi, come per esempio Python.

Ora possiamo avviare il codice, ci basterà dare il comando “.” seguito dal nome del file compilato che abbiamo scelto prima.
Nel nostro caso sarà “**./BOF**”.



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

(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:
```



Facciamo un test per vedere come risponde il codice originale.

Inseriamo un nome più lungo dei 10 caratteri consentiti, come vedete nello screen riportato qui sotto ci verrà riportato un errore di **“segmentation fault”**, questo accade quando il programma tenta di scrivere in una posizione di memoria a lui non consentita.

```
(kali㉿kali)-[~/Desktop]  
└ $ ./BOF
```

```
Si prega di inserire il nome utente:alasksncjebnjwncjsanbcjbsabesabac  
Nome utente inserito: alasksncjebnjwncjsanbcjbsabesabac  
zsh: segmentation fault (core dumped) ./BOF
```



Proviamo allora ad apportare una semplice modifica.

Andremo a specificare, nella riga “**scanf**”, che l’input che l’utente inserirà, e che di seguito verrà stampato a schermo, avrà una lunghezza massima di 10 caratteri



```
GNU nano 7.2 7.2                                BOF.c.c
#include <stdio.h>
// Includiamo string.h // Inseriamo una ulteriore
int main () {
    int main() {
char buffer[10];
    }
    has
        printf("Si prega di inserire il nome utente:");
printf ("Si prega di inserire il nome utente:");
scanf ("%10s", buffer);
    size_t length = strlen(buffer); // Se pre
printf ("Nome utente inserito: %10s\n", buffer);
    buffer[length - 1] = '\0';
}
return 0;
}
printf("Nome utente inserito: %s\n", buffer);
}
return 0;
```

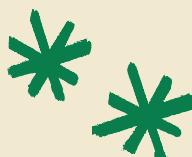


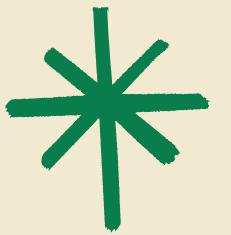


Come possiamo vedere nell'immagine sottostante, adesso il codice stampa solo i primi 10 caratteri dell'input utente.

Come noterete se l'input risulta più lungo del dovuto, dal decimo carattere in poi non viene stampato più nulla, comunque il codice non ci dà nessun messaggio di errore.

```
(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:FrancescoBianchi
Nome utente inserito: FrancescoB
```

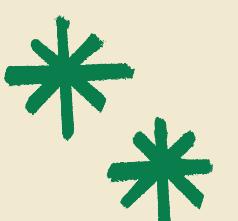
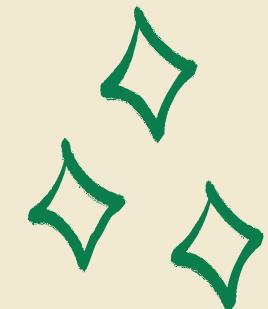




```
GNU nano 7.2 7.2                                BOF.c *
```

```
#include <stdio.h>
// #include <string.h> //Inseriamo una ulteriore linea di codice per testare l'overflow
int main () {
    int main() {
char buffer [30];buffer[10];
    }
    printf("Si prega di inserire il nome utente:");
    printf ("Si prega di inserire il nome utente:");
    scanf ("%30s", buffer);
    size_t length = strlen(buffer); // Se presento un overflow, la lunghezza del buffer non è più 10
    printf ("Nome utente inserito: %30s\n", buffer);
    buffer[length - 1] = '\0';
}
return 0;
}
printf("Nome utente inserito: %s\n", buffer);
return 0;
```

Proviamo allora ad aumentare la dimensione del **buffer** questa volta a **30**, come consigliato dalla slide, e testiamo nuovamente il codice.



Come nel caso precedente il codice stampa correttamente i primi 30 caratteri ed evita di riportare il resto dell'input dell'utente se appunto supera questo valore.

Questi due sono dei classici casi di **Buffer Overflow**, dove il codice non implementa dei controlli di sicurezza nel campo dove l'utente può scrivere quello che vuole per esempio un codice malevolo, immaginate le possibili conseguenze se per caso fossimo su un forum o un social network invece di un ambiente virtuale.

Anche se molto improbabile che un utente scelga un nome lungo 30 caratteri, questo non basterà come controllo di sicurezza, ho dunque apportato delle modifiche a questo codice.

```
(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:FrancescoClaudioAndreaFedericoGianlucaPierGiovanni
Nome utente inserito: FrancescoClaudioAndreaFederico
```



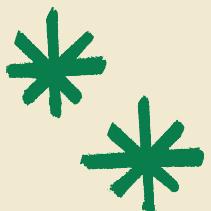
Dunque, prima abbiamo aumentato la dimensione del buffer da 10 a 30 caratteri per consentire l'inserimento di un input più lungo.

Dopodichè, abbiamo introdotto un ciclo **do-while** per gestire le condizioni relative alla lunghezza dell'input inserito dall'utente. Questo ciclo continuerà a richiedere all'utente l'inserimento finché non lo accetterà, ciò vuol dire un input di 30 caratteri o inferiore.

All'interno del ciclo, abbiamo utilizzato la funzione **fgets** al posto di **scanf**.

Fgets migliora la robustezza del programma in quanto a differenza di **scanf** controlla meglio la stringa anche in caso di inserimento di spazi tra un carattere e l'altro.

In aggiunta abbiamo anche due righe di codice per **pulire il buffer** dell'input, così da evitare che i caratteri overflow precedenti fossero presi in automatico dal programma per la nuova richiesta del nome utente.



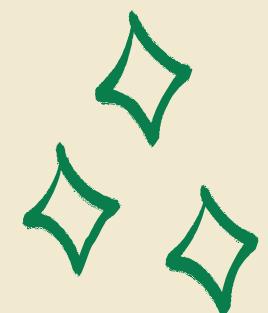
Riporto il codice nella slide successiva.

```
int main() {  
  
    char buffer[31];  
  
    do {  
  
        printf("Si prega di inserire il nome utente:\n");  
        fgets(buffer, sizeof(buffer), stdin);  
  
        // Pulisce il buffer di input  
        int c;  
        while ((c = getchar()) != '\n' && c != EOF);  
  
        // Rimuove il carattere di nuova linea finale inserito da fgets  
        buffer[strcspn(buffer, "\n")] = '\0';  
  
        if (strlen(buffer) ≥ 30) {  
            printf("Puoi inserire massimo 30 caratteri\n");  
        } else {  
            printf("Nome utente inserito: %s\n", buffer);  
        }  
  
    } while (strlen(buffer) ≥ 30);  
  
    return 0;  
}
```

Avviamo nuovamente il codice e vediamo come risponde questa volta.



Come noterete nell'immagine in calce, a differenza dei primi tentativi, ora il codice chiederà di immettere di nuovo il nome utente se l'input utente sarà più lungo dei 30 caratteri consentiti.



```
(kali㉿kali)-[~/Desktop]
$ ./BOF
Si prega di inserire il nome utente:
123456789012345678901234567890mmm
Puoi inserire massimo 30 caratteri
Si prega di inserire il nome utente:
|
```





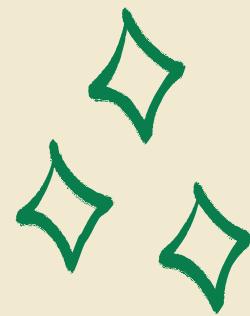
Non ci resta che modificare il codice ancora una volta per stampare a schermo gli indirizzi di dove effettivamente verranno stampati i caratteri in overflow.
Quindi ho infine introdotto istruzioni di stampa (printf) aggiuntive nel codice.

Di seguito le righe di codice aggiunte al precedente:

“printf(*“Caratteri in overflow: %s\n”*, buffer + 30);”

“printf(*“Posizione in memoria dei caratteri di overflow: %p\n”*, (void*)(buffer + 30));”

La slide successiva mostra l’aggiunta di queste righe di codice.



```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[31];

    do {
        // Ci dice dove viene memorizzato la nostra variabile buffer (serve da confronto)
        printf("Indirizzo di memoria del buffer: %p\n", (void*)buffer);

        printf("Si prega di inserire il nome utente:\n");
        fgets(buffer, sizeof(buffer), stdin);

        // Pulisce il buffer di input
        int c;
        while ((c = getchar()) != '\n' && c != EOF);

        // Rimuove il carattere di nuova linea finale inserito da fgets
        buffer[strcspn(buffer, "\n")] = '\0';

        if (strlen(buffer) >= 30) {
            printf("Puoi inserire massimo 30 caratteri\n");
            printf("Caratteri in overflow: %s\n", buffer + 30);
            printf("Posizione in memoria dei caratteri di overflow: %p\n", (void*)(buffer + 30));
        } else {
            printf("Nome utente inserito: %s\n", buffer);
        }

    } while (strlen(buffer) >= 30);

    return 0;
}
```

Nell'immagine sottostante possiamo finalmente vedere che il codice modificato riporterà gli indirizzi di memoria di dove verranno scritti i caratteri.

Questo è un ottimo punto di partenza, in caso ci trovassimo di fronte ad una vulnerabilità del genere, per impostare delle **remediation action** e far fronte così a questo problema.

```
(kali㉿kali)-[~/Desktop]
$ ./BOF
Indirizzo di memoria del buffer: 0x7ffe1b752a0
Si prega di inserire il nome utente:
123456789012345678901234567890mmm
Puoi inserire massimo 30 caratteri
Caratteri in overflow:
Posizione in memoria dei caratteri di overflow: 0x7ffe1b752be
Indirizzo di memoria del buffer: 0x7ffe1b752a0
Si prega di inserire il nome utente:
█
```