Università di Catania

Binary exploitation

Progetto internet security 2022/2023

Marco Lo Bello
Matricola: 1000016159
Email: marcolobello.sir@gmail.com

Indice

Introduzione....2

Memoria di un processo....2

Perché usare la stack....3

Stack Region....3

Buffer overflow....5

- *Esempio di buffer overflow....5*

Binary exploitation....6

Esempio reale: CVE 2021-3156 (Baron Samedit)....8

- *Da cosa era causato il buffer overflow?....8*
- *Come è stata risolta la vulnerabilità?....12*

Come prevenire un buffer overflow....13

Conclusioni....13

Riferimenti....13

Introduzione

In questo documento verrà illustrato come poter sfruttare una vulnerabilità presente in Ubuntu 20.04, che consentiva di effettuare privilege escalation da utente generico, ad utente root tramite un attacco di tipo **binary exploiation.**

Nella prima parte del documento saranno spiegati alcuni concetti e mostrati alcuni esempi per comprendere meglio cos'è il binary exploitation.

Nella seconda parte invece, verrà dimostrato l'attacco analizzando come ciò fosse possibile e come è stato risolto.

Nella terza parte verranno infine presentati dei piccoli consigli per prevenire un errore di tipo buffer overflow.

Memoria di un processo

Un processo è composto da:

- **Text**: contiene il codice del programma.
- **Data**:
  - Variabili globali in memoria centrale (data)
  - Variabili temporanee (stack)
  - Variabili allocate dinamicamente (heap)
- **Stack**: è un tipo di dato astratto, utilizzato per gestire le informazioni. In una stack l'ultimo oggetto inserito deve essere il primo a essere rimosso, questa proprietà è chiamata LIFO (Last In First Out). Per garantire la proprietà sulla stack si possono essere effettuate solo due operazioni:
  - **Push:** permette di inserire un elemento in cima alla stack
  - **POP:** permette di rimuovere l'elemento in cima alla stack

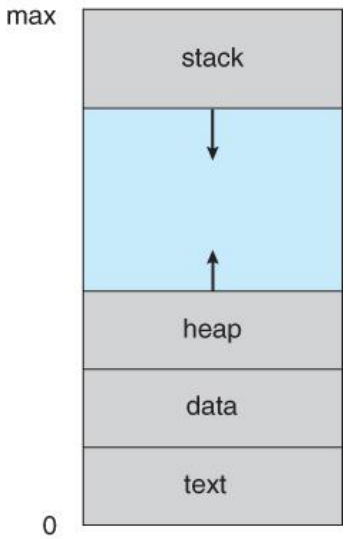

Figura 1: organizzazione memoria processo

Perché usare la stack

Al giorno d'oggi si utilizzano per lo più linguaggi ad alto livello. Bisogna ricordare però che comunque questi linguaggi sono delle astrazioni di un linguaggio a basso livello, come l'assembly.

La tecnica più importante per supportare queste serie di astrazioni e strutturare al meglio i programmi realizzati con linguaggi ad alto livello è la tecnica della chiamata a funzione.

Quando viene effettuata una chiamata a una funzione, si modifica il flusso di esecuzione del programma, perché al posto di eseguire la riga successiva, si effettua un salto alla funzione chiamata.

La differenza ricade però tra l'istruzione JMP del linguaggio assembly e quella di chiamata a funzione; infatti, quest'ultima non obbliga a sapere l'indirizzo di memoria da dove riprendere il flusso perché, quando la funzione finisce, il programma riprenderà la sua esecuzione dalla riga successiva alla chiamata.

Questa astrazione viene implementata proprio tramite la stack.

Stack Region

Per utilizzare la stack, vengono utilizzati dei registri. Il registro principale è l'SP (Stack Pointer) che contiene l'indirizzo di memora della cima della stack.

La struttura della stack si può decomporre in una serie di regioni denominate stack frame, i quali vengono creati durante la chiamata a una funzione, ed eliminati al termine dell'esecuzione.

Uno stack frame contiene varie informazioni che aiutano il processore a eseguire la funzione del programma, in particolare contiene: i parametri della funzione, le variabili locali della funzione e i dati necessari per recuperare lo stack frame della funzione precedente. Tra questi dati vi è l'IP (Instruction Pointer), che indica l'indirizzo in cui vi è la successiva istruzione da eseguire.

Oltre ad avere il registro SP, è molto utile avere un altro registro chiamato FP (Frame Pointer) che punta lo stack frame attualmente in uso. In linea di principio le variabili locali potrebbero essere referenziate fornendo l'offset a partire da SP. Tuttavia, durante l'esecuzione, l'indirizzo di SP varia e questo rende il calcolo degli offset poco pratico.

Quindi per migliorare le prestazioni viene utilizzato il registro FP, che puntando ad un'area fissa, permette di fissare l'offset e di non ricalcolarlo a seguito delle operazioni di Push e di POP.

Una cosa importante da ricordare è che, quando si chiama una funzione, la prima operazione che viene effettuata è una routine che consiste nelle seguenti operazioni:

1. Salvare il valore del FP.
2. Spostare FP in modo tale da farlo puntare alla base della nuova regione.
3. Incrementare la dimensione di SP per riservare spazio per le variabili locali.

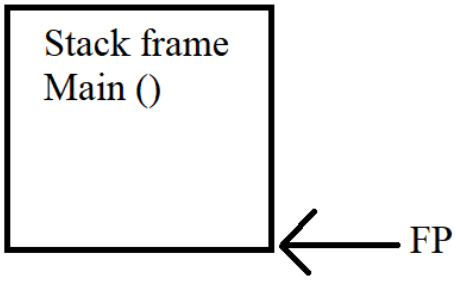

Figura 2: stack frame iniziale

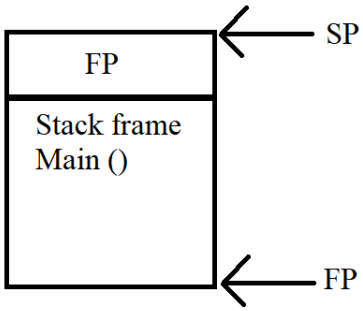

Figura 3: passo 1

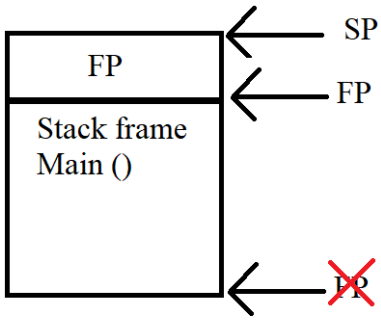

Figura 4: passo 2

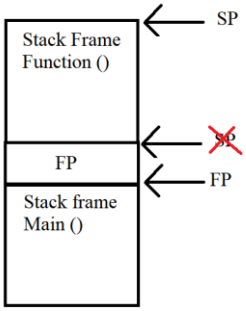

Figura 5: passo 3

Buffer overflow

Un errore di buffer overflow si verifica quando a un programma viene fornito una quantità di dati maggiore rispetto alla sua effettiva capacità di memoria. Come conseguenza, questo potrebbe segnalare errori o comportarsi in modo anomalo.

Le informazioni in eccesso, di conseguenza, vanno a finire in posizioni di memoria adiacenti, corrompendo o sovrascrivendo i dati presenti, nel caso in cui quegli indirizzi siano assegnati al programma.

- Esempio di buffer overflow

```
 5 void function(char *str) {
 6    char buffer[16];
 7
 8    strcpy(buffer,str);
 9 }
10
11 void main() {
12   char large_string[256];
13   int i;
14
15   for( i = 0; i < 255; i++)
16     large_string[i] = 'A';
17
18   function(large_string);
19 }
```

Figura 6: esempio di buffer overflow

Questo codice genere un errore di **segmentazione,** perché all'interno del buffer *large_string* vengono inserite 256 'A'. Dopo viene chiamata la funzione *strcpy* che copia le lettere 'A' nel buffer *buffer* la cui dimensione però non è 256, bensì 16. L'array b*uffer,* essendo più piccolo non riuscirà a contenere tutte le lettere 'A' causando quindi un **buffer overflow.**

Dopo aver eseguito *function* la memoria viene sovrascritta da tutti i caratteri 'A', di conseguenza verrà perso l'indirizzo di ritorno e quindi il programma non potrà continuare la sua normale esecuzione.

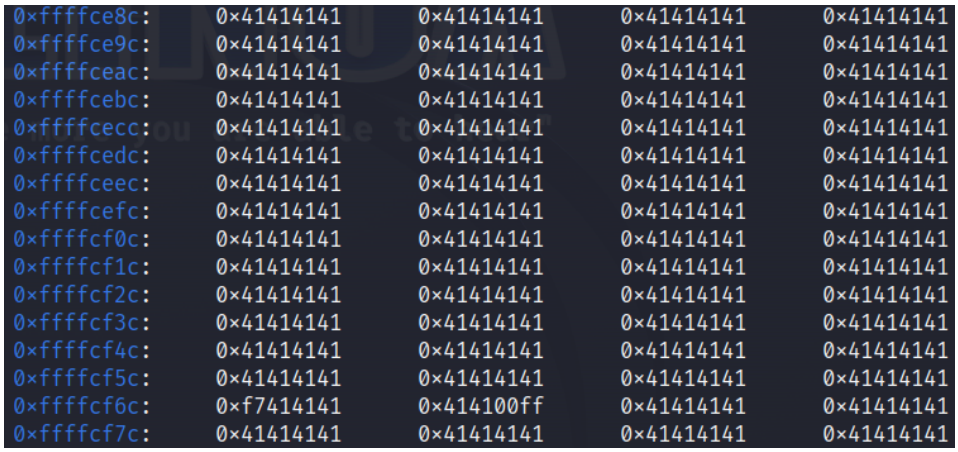

Figura 7: memoria processo

Binary exploitation

Il binary exploitation consiste nello sfruttare una vulnerabilità del codice al fine di causare un comportamento inaspettato nel programma.

```
 5 void function(int a, int b, int c) {
 6    char buffer[16];
 7    int *ret;
 8
 9 }
10
11 void main() {
12   int x;
13
14   x = 0;
15   function(1,2,3);
16   x = 1;
17   printf("%d\n",x);
18 }
```

Figura 8: codice da attaccare

In questo codice, è presente una variabile *x* che viene messa a 0, dopo viene chiamata la funzione *function* che non fa nulla se non allocare un buffer e un puntatore intero e in fine, si torna nel main dove *x* diventa 1 e viene stampato il risultato.

Come potremmo manipolare l'esecuzione del codice, in modo tale da far saltare l'esecuzione della riga 16 a far stampare 0 e non il valore 1 alla fine del programma?

Il codice riportato di seguito permette questo:

```
 5 void function(int a, int b, int c) {
 6    char buffer[16];
 7    int *ret;
 8
 9    memset(buffer, 'A',5);
10
11    ret = buffer + 4*8;
12
13    (*ret) = *ret + 10;
14 }
15
16 void main() {
17   int x;
18
19   x = 0;
20   function(1,2,3);
21   x = 1;
22   printf("%d\n",x);
23 }
```

Figura 9: esempio di binary exploitation

Avendo aggiunto le righe 9, 11 e 13 andiamo a modificare l'indirizzo di ritorno del main, quindi quando la funzione *function* terminerà l'esecuzione del programma non riprenderà dalla riga 21, ma ripartirà dalla 22, andando a stampare 0 al posto di 1.

Quello che succede nel dettaglio è questo:

```
0x5655620c <+45>:    call   0x5655619d <function>
0x56556211 <+50>:    add    $0x10,%esp
```

Figura 10: indirizzo di ritorno

L'indirizzo 0x56556211 è l'indirizzo da dove il programma deve riprendere dopo la chiamata a *function*.

```
0xffffcf50:    0xffffcf6c    0x00000041    0x00000005    0x565561a9
0xffffcf60:    0xffffffff    0xf7fc9694    0xf7ffd608    0x00000000
0xffffcf70:    0xf7ffcff4    0x0000002c    0x00000000    0xffffdfce
0xffffcf80:    0xf7fc7550    0x56558ff4    0xffffcfb8    0x56556211
0xffffcf90:    0x00000001    0x00000002    0x00000003    0x565561f6
0xffffcfa0:    0xffffcfe0    0xf7fc1678    0xf7fc1b50    0x00000000
0xffffcfb0:    0xffffcfd0    0xf7e1cff4    0x00000000    0xf7c23295
0xffffcfc0:    0x00000000    0x00000070    0xf7ffcff4    0xf7c23295
```

Figura 11: indirizzo di ritorno in memoria

Tramite le istruzioni aggiunte noi andiamo a cambiare questo indirizzo, e quando prima dell'uscita di function il programma avrà in memoria un altro indirizzo.

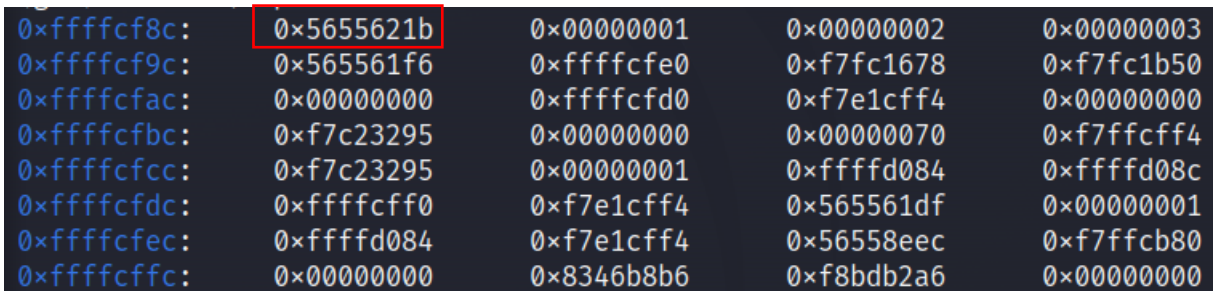

Figura 12: indirizzo di ritorno cambiato

L'indirizzo salvato è lo **0x5655621b** che viene dopo l'indirizzo di prima **0x56556211.** In particolare, noi abbiamo saltato l'istruzione *x=1* e quindi il programma stamperà 0.

In uno scenario reale è quasi impossibile che un attaccante vada a scrivere righe di codice all'interno di un programma, però abbiamo visto in precedenza che a seguito di un buffer overflow si può riempire la stack region di un programma. Questo è come viene realizzato il binary exploitation, sfruttando una vulnerabilità del programma, come per esempio un buffer overflow per andare a modificare il flusso di esecuzione.

Esempio reale: CVE 2021-3156 (Baron Samedit)

Le versioni di sudo dalla 1.8.2 alla 1.8.31, soffrivano di una vulnerabilità basata sull'heap overflow che consentiva agli utenti senza privilegi di ottenere i privilegi root.

Da cosa era causato il buffer overflow?

```
1  if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
2      char **av, *cmnd = NULL;
3      int ac = 1;
4
5      // [...]
6
7      for (av = argv; *av != NULL; av++) {
8          for (src = *av; *src != '\0'; src++) {
9          /* quote potential meta characters */
10         if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
11             *dst++ = '\\';
12         *dst++ = *src;
13         }
14         *dst++ = ' ';
15     }
```

Figura 13: codice sudo_1

Quando su sudo viene eseguito un comando in un terminale, la riga del comando inserita viene concatenata (7 – 15) eliminando tutti i caratteri speciali (10 – 11).

```
1  for (size = 0, av = NewArgv + 1; *av; av++)
2      size += strlen(*av) + 1;
3
4  if (size == 0 || (user_args = malloc(size)) == NULL) {
5      sudo_warnx(U_("%s: %s"), __func__, U_("unable to allocate memory"));
6      debug_return_int(-1);
7  }
```

Figura 14: codice sudo_2

Dopo gli argomenti della riga di comando vengono concatenati e il buffer *user_args* viene allocato nell'heap.

```
1  if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
2      for (size = 0, av = NewArgv + 1; *av; av++)
3          size += strlen(*av) + 1;
4          if (size == 0 || (user_args = malloc(size)) == NULL) {
5      }
6      //...
7  }
8
9  if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
10     for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
11         while (*from) {
12             if (from[0] == '\\' && !isspace((unsigned char)from[1]))
13                 from++;
14             *to++ = *from++;
15         }
16         *to++ = ' ';
17     }
18     *--to = '\0';
19
```

Figura 15: codice sudo_3

Successivamente la stringa viene copiata carattere per carattere nel buffer (9-18).

Questo buffer sarà quello vulnerabile al buffer overflow.

Se la condizione dell'*if* (12) è vera, viene spostato di una posizione in avanti il puntatore *from*. Facendo ciò, verrà copiato il carattere successivo, dato che le stringhe in C terminano con *null* e questo ciclo (11) continua a copiare fin quando non c'è un byte *null* nella stringa, se la stringa finisce con il carattere '\' "backslash", prima del *null* byte il carattere successivo che verrà copiato sarà proprio *null.* Verificandosi la condizione dell'*if,* il puntatore *from* verrà spostato avanti.

Una volta che viene spostato avanti il puntatore, si verificherà il buffer overflow nel buffer *user_args* se si continuerà a riempirlo.

```
if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
    char **av, *cmnd = NULL;
    int ac = 1;

// [...]

    for (av = argv; *av != NULL; av++) {
        for (src = *av; *src != '\0'; src++) {
        /* quote potential meta characters */
        if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
            *dst++ = '\\';
        *dst++ = *src;
        }
        *dst++ = ' ';
    }
```

Figura 16: codice sudo_4

Ovviamente gli sviluppatori avevano pensato a questa problematica, di fatto, prima di passare alla funzione vista prima (figura 11), il codice passa prima da un'altra funzione (figura 12) che serve proprio a sanificare l'input. Infatti, se per esempio il comando inserito termina con un backslash, la funzione ne aggiunge un altro (7- 15), in modo tale da far funzionare l'*if* visto prima.

La cosa strana è che la sanificazione non viene effettuata direttamente sulla prima funzione, ma che la funzione si debba fidare dell'input mandato dalla seconda. Se ci fosse un modo per bypassare la seconda funzione a arrivare direttamente alla prima, si potrebbe scrivere la stringa con il backslash e di conseguenza creare il buffer overflow.

Sudo utilizza dei flag che consentono di capire se stiamo scrivendo comandi nel terminale. Per arrivare alla funzione vulnerabile esistono due modi:

1. Attivare uno dei flag tra *MODE_RUN, MODE_EDIT* o *MODE_CHECK* (figura 11)
2. Attivare uno dei flag tra *MODE_SHELL* oppure *MODE_LOGIN_SHELL* (figura 11)

Mentre per raggiungere la funzione che sanifica l'input esiste solo un modo ed è quello di avere i flag *MODE_RUN* e *MODE_SHELL* contemporaneamente attivi.

Quindi se ci fosse un modo per attivare i flag *MODE_SHELL* o *MODE_EDIT* o *MODE_CHECK* ma avere disattivato il flag *MODE_RUN* noi potremmo raggiungere la sezione di codice vulnerabile senza passare dalla funzione sanificatrice.

Apparentemente sembrava che questo caso non potesse verificarsi perché, se uno tra i flag *MODE_SET* (361) oppure *MODE_CHECK* erano attivi (423- 519), il flag *MODE_SHELL* veniva rimosso da *valid_flags* (363 e 424), inoltre l'esecuzione terminava con un messaggio di errore in caso venisse impostato un altro flag come, per esempio, *MODE_EDIT* o *MODE_CHECK* che potevano servire a raggiungere il codice vulnerabile.

```
358                 case 'e':
...
361                     mode = MODE_EDIT;
362                     sudo_settings[ARG_SUDOEDIT].value = "true";
363                     valid_flags = MODE_NONINTERACTIVE;
364                     break;
...
416                 case 'l':
...
423                     mode = MODE_LIST;
424                     valid_flags = MODE_NONINTERACTIVE|MODE_LONG_LIST;
425                     break;
...
518     if (argc > 0 && mode == MODE_LIST)
519         mode = MODE_CHECK;
...
532     if ((flags & valid_flags) != flags)
533         usage(1);
```

Figura 17: flag

C'era però un problema, in caso fosse stato eseguito sudo come "*sudoedit*" al posto della normale esecuzione "*sudo*", sarebbe stato attivato il flag *MODE_EDIT* (270) senza resettare i *valid_flags* di cui *MODE_SHELL* faceva parte (127- 249).

```
127 #define DEFAULT_VALID_FLAGS
(MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)
...
249     int valid_flags = DEFAULT_VALID_FLAGS;
...
267     proglen = strlen(progname);
268     if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
269         progname = "sudoedit";
270         mode = MODE_EDIT;
271         sudo_settings[ARG_SUDOEDIT].value = "true";
272     }
```

Figura 18: sudoedit

Di conseguenza, scrivendo il comando "*sudoedit -s*" venivano attivati i flag *MODE_EDIT* e *MODE_SHELL* ma non veniva attivato *MODE_RUN*, permettendo quindi di raggiungere il codice vulnerabile.

Come è stata risolta la vulnerabilità?

La vulnerabilità è stata risolta apportando le seguenti modifiche al codice:

```
2.16            progname = "sudoedit";
2.17            mode = MODE_EDIT;
2.18            sudo_settings[ARG_SUDOEDIT].value = "true";
2.19    +       valid_flags = EDIT_VALID_FLAGS;
2.20        }
```

Figura 19: sudoedit fix

La prima modifica è stata quella di resettare i *valid_flags* (2.19) quando si va in "*sudoedit*".

L'altra modifica è stata effettuata alla funzione vulnerabile.

```
1.35                for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
1.36                    while (*from) {
1.37    -                   if (from[0] == '\\' && !isspace((unsigned char)from[1]))
1.38    +                   if (from[0] == '\\' && from[1] != '\0' &&
1.39    +                           !isspace((unsigned char)from[1])) {
1.40                            from++;
1.41    +                   }
1.42    +                   if (size - (to - user_args) < 1) {
1.43    +                       sudo_warnx(U_("internal error, %s overflow"),
1.44    +                           __func__);
1.45    +                       debug_return_int(NOT_FOUND_ERROR);
1.46    +                   }
1.47                        *to++ = *from++;
1.48                    }
1.49    +               if (size - (to - user_args) < 1) {
1.50    +                   sudo_warnx(U_("internal error, %s overflow"),
1.51    +                       __func__);
1.52    +                   debug_return_int(NOT_FOUND_ERROR);
1.53    +               }
1.54                    *to++ = ' ';
1.55                }
1.56                *--to = '\0';
```

Figura 20: funzione vulnerabile fix

Il vecchio if è cambiato (1.38) e adesso sposta avanti il puntatore solo se vengono soddisfatte queste condizioni:

- La stringa deve iniziare con '\\'
- Il secondo carattere della stringa non deve essere '\0'
- Il secondo carattere della stringa non deve essere uno spazio.

Inoltre, sono stati inseriti due *if* (1.42 e 1.49) che controllano se eventuali stringhe vanno fuori dalla dimensione (*size*) della stringa fornita come argomento.

Come prevenire un buffer overflow

Scoprire una vulnerabilità basata sul buffer overflow non sempre è semplice, basti pensare che la vulnerabilità di sudo mostrata prima è stata scoperta dopo 10 anni da quando è stata erroneamente introdotta.

Di seguito vengono riportati dei consigli che servono a evitare problemi di buffer overflow:

- Controllo del codice, sia manualmente che automatizzato;
- Usare compilatori appositi;
- Utilizzare funzioni sicure, come per esempio *strncpy* al posto di *strcpy*;
- Fuzzing: è una tecnica automatizzata che consiste nel monitorare il programma per rilevare errori mentre vengono inseriti degli input mal formati durante l'esecuzione.

Conclusioni

È stato spiegato come è organizzata la memoria di un processo, grazie ai due esempi è stato possibile capire come questa vulnerabilità può essere sfruttata per realizzare degli attacchi di binary exlpoitation.

Inoltre, la dimostrazione ci ha fatto vedere come in un sistema reale un errore di buffer overflow possa essere veramente un problema da non sottovalutare.

Infine, sono state descritte delle semplici tecniche per aiutare a prevenire un errore di buffer overflow.

Riferimenti

Slide del corso sistemi operativi unibg:
https://cs.unibg.it/gherardi/so2013/slides/03.pdf

Qualys Community: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

LiveOverflow: https://liveoverflow.com/critical-sudo-vulnerability-walkthrough-cve-2021-3156/

Repository Sudo: https://www.sudo.ws/repos/sudo

Repository Exploit per la dimostrazione: https://github.com/CptGibbon/CVE-2021-3156

Wikipedia (fuzzing): https://it.wikipedia.org/wiki/Fuzzing