VULNERABILITĂȚI ÎN MANAGEMENTUL MEMORIEI

Programele C și C++ care operează pe date cu o dimensiune variabilă necesită alocarea dinamică a memoriei pentru a gestiona aceste date. Marea majoritate a aplicațiilor care nu sunt critice pentru siguranță utilizează alocarea dinamică a memoriei. Memoria heap este gestionată de un alocator dinamic de memorie sau de un manager de memorie. Deși detaliile despre modul în care acești manageri de memorie sunt exploatați variază, toate aceste vulnerabilități rezultă dintr-un set restrâns de comportamente nedefinite care sunt introduse în program din cauza erorilor de codare date de gestionarea memoriei de către programator.

Funcțiile de gestionare a memoriei sunt specificate de standardul C și sunt disponibile în implementările compilatoarelor existente pe mai multe platforme.

Funcția malloc (size_t size) alocă size octeți și returnează un pointer către memoria alocată. Memoria alocată nu este inițializată la o valoare cunoscută.

Funcția realloc (void *p, size_t size) modifică dimensiunea blocului de memorie indicat de p la size octeți. Memoria nou alocată va fi neinițializată și, în consecință, va avea valori nedeterminate. Dacă solicitarea de memorie nu poate fi efectuată cu succes, obiectul vechi este lăsat intact și nu se modifică valori. Dacă p este un pointer nul, apelul este echivalent cu malloc(size). Dacă dimensiunea este egală cu 0, apelul este echivalent cu free(p), cu precizarea că acest mod de eliberare a memoriei ar trebui evitat.

Funcția calloc(size_t nmemb, size_t size) alocă memorie pentru nmemb elemente de size octeți fiecare și returnează un pointer către memoria alocată. Memoria este inițializată cu 0.

Funcția de dezalocare a memoriei free(void *p) eliberează spațiul de memorie indicat de p, care a fost returnat printr-un apel anterior la malloc(), calloc() sau realloc().

Comportamentul nedefinit apare dacă memoria referențiată nu a fost alocată de una dintre aceste funcții sau dacă free(p) a fost apelat anterior. Dacă p este un pointer nul, nu se efectuează nicio operație.

PRESUPUNEREA INCORECTĂ CĂ MEMORIA A FOST INIȚIALIZATĂ

Funcția malloc() este frecvent utilizată pentru a aloca blocuri de memorie. Valorile spațiului returnat de malloc() sunt nedeterminate. O eroare comună este presupunerea incorectă că malloc() inițializează toți biții cu zero.

În exemplu, instrucțiunea de atribuire de pe linia 36 presupune că valoarea lui array2[i] este inițial 0, iar funcția returnează un rezultat incorect.

```
19  int main() {
20
21      populateMemoryWithSomeData();
22
23      int size = 10;
24
25      int *array1 = (int *)malloc(size * sizeof(int));
26      int *array2 = (int *)malloc(size * sizeof(int));
27
28      if (array1 == NULL || array2 == NULL) {
29          printf("Memory allocation failed.\n");
30          free(array1);
31          free(array2);
32          return 1;
33      }
34
35      for (int i = 0; i < size; i++) {
36          array2[i] = i * 2;
37          array1[i] += array2[i];
38          printf("%d ", array1[i]);
39      }
40
41      printf("%s", "\n");
42
43      free(array1);
44      free(array2);
45
46      return 0;
47  }
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/ReadingUninitialized
Memory$ gcc -ggdb -o IncorrectInitializationAssumptionBefore IncorrectInitializationAssumptio
nBefore.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/ReadingUninitialized
Memory$ ./IncorrectInitializationAssumptionBefore
1000 1003 1006 1009 1012 1015 1018 1021 1024 1027
```

Inițializarea blocurilor mari de memorie poate degrada performanța și nu este întotdeauna necesară. Decizia comitetului de standarde C de a nu solicita malloc() să inițializeze această memorie rezervă această decizie programatorului. Dacă este necesar, se poate inițializa memoria utilizând memset() sau apelând calloc(), care zeroizează memoria. Pentru a rezolva această problemă, apelul malloc() a fost înlocuit cu un apel calloc() pentru ca memoria să fie setată cu 0.

```c
int main() {

    populateMemoryWithSomeData();

    int size = 10;

    int *array1 = (int *)calloc(size, sizeof(int));
    int *array2 = (int *)calloc(size, sizeof(int));

    if (array1 == NULL || array2 == NULL) {
        printf("Memory allocation failed.\n");
        free(array1);
        free(array2);
        return 1;
    }

    for (int i = 0; i < size; i++) {
        array2[i] = i * 2;
        array1[i] += array2[i];
        printf("%d ", array1[i]);
    }

    printf("%s", "\n");

    free(array1);
    free(array2);

    return 0;
}
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/ReadingUninitialized
Memory$ gcc -ggdb -o IncorrectInitializationAssumptionAfter IncorrectInitializationAssumption
After.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/ReadingUninitialized
Memory$ ./IncorrectInitializationAssumptionAfter
0 2 4 6 8 10 12 14 16 18
```

LIPSA SUPRASCRIERII MEMORIEI DUPĂ OPERAȚII CU DATE SENSIBILE

Eșecul inițializării memoriei poate crea, de asemenea, un risc asupra confidențialității. Fragmentele incluse ale fișierului /etc/passwd sunt un exemplu de scurgere de informații care ar putea compromite securitatea sistemului.

```
5   void readSensitiveDataFrom(const char *filename)
6   {
7       FILE *file = fopen(filename, "rb");
8       if (file == NULL)
9       {
10          perror("Error opening file");
11          return;
12      }
13
14      fseek(file, 0, SEEK_END);
15      long length = ftell(file);
16      rewind(file);
17
18      char *buffer = malloc(length);
19      if (buffer == NULL)
20      {
21          perror("Memory allocation failed");
22          fclose(file);
23          return;
24      }
25
26      if (fread(buffer, 1, length, file) != length)
27      {
28          perror("Error reading file");
29          fclose(file);
30          free(buffer);
31          return;
32      }
33
34      free(buffer);
35      fclose(file);
36  }
```

```c
void exposeHeapDataTo(const char *filename)
{
    size_t bufferSize = 1024 * sizeof(char);

    char *buffer = (char *)malloc(bufferSize);
    if (buffer == NULL)
    {
        perror("Failed to allocate memory");
        return;
    }

    strcpy(buffer, "Pretend to read some data into the buffer");

    FILE *file = fopen(filename, "wb");
    if (file == NULL)
    {
        perror("Error opening file");
        free(buffer);
        return;
    }

    if (fwrite(buffer, 1, bufferSize, file) != bufferSize)
    {
        perror("Error writing to file");
    }

    free(buffer);
    fclose(file);
}

int main()
{
    readSensitiveDataFrom("/etc/passwd");
    printf("%s", "The memory was deallocated but the sesitive data still
    exposeHeapDataTo("output.bin");
    printf("%s", "Whoops! Some data got leaked...\n");
    return 0;
}
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities$ gcc -ggdb -o Read
ingUninitializedMemoryBefore ReadingUninitializedMemoryBefore.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities$ ./ReadingUninitia
lizedMemoryBefore
The memory was deallocated but the sesitive data still resides in the heap mwahahaha
Whoops! Some data got leaked...
```

output.bin

```
1   Pretend to read some data into the buffer nologin
2   systemd-timesync:x:103:106:systemd Time Synchronization,,,:/run/systemd:,
3   syslog:x:104:111::/home/syslog:/usr/sbin/nologin
4   _apt:x:105:65534::/nonexistent:/usr/sbin/nologin
5   tss:x:106:113:TPM software stack,,,:/var/lib/tpm:/bin/false
6   uuidd:x:107:116::/run/uuidd:/usr/sbin/nologin
7   systemd-oom:x:108:117:systemd Userspace OOM Killer,,,:/run/systemd:/usr/s
8   tcpdump:x:109:118::/nonexistent:/usr/sbin/nologin
9   avahi-autoipd:x:110:119:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/us
10  usbmux:x:111:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
11  dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
12  kernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
13  avahi:x:114:121:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin
14  cups-pk-helper:x:115:122:user for cups-pk-helper service,,,:/home/cups-pk
15  rtkit:x:116:123:RealtimeKit,,,:/proc:/usr/sbin/nologin
16  whoopsie:x:117:124::/nonexis
```

```
denissa@denissa-vm:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
systemd-timesync:x:103:106:systemd Time Synchronization,,,:/run/systemd:/usr/sbin
/nologin
syslog:x:104:111::/home/syslog:/usr/sbin/nologin
_apt:x:105:65534::/nonexistent:/usr/sbin/nologin
tss:x:106:113:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:116::/run/uuidd:/usr/sbin/nologin
systemd-oom:x:108:117:systemd Userspace OOM Killer,,,:/run/systemd:/usr/sbin/nolo
gin
tcpdump:x:109:118::/nonexistent:/usr/sbin/nologin
avahi-autoipd:x:110:119:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/n
ologin
usbmux:x:111:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
kernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
avahi:x:114:121:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin
cups-pk-helper:x:115:122:user for cups-pk-helper service,,,:/home/cups-pk-helper:
/usr/sbin/nologin
rtkit:x:116:123:RealtimeKit,,,:/proc:/usr/sbin/nologin
whoopsie:x:117:124::/nonexistent:/bin/false
sssd:x:118:125:SSSD system user,,,:/var/lib/sss:/usr/sbin/nologin
speech-dispatcher:x:119:29:Speech Dispatcher,,,:/run/speech-dispatcher:/bin/false
fwupd-refresh:x:120:126:fwupd-refresh user,,,:/run/systemd:/usr/sbin/nologin
```

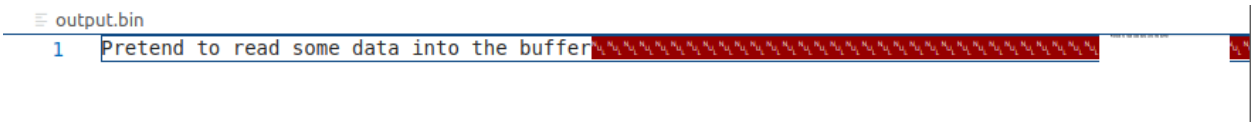
Problema în acest caz este că programul nu inițializează memoria alocată dinamic pe care o folosește pentru a citi un bloc de date de pe disc. Înainte de a aloca acest bloc, programul a invocat funcția readSensitiveDataFrom() pentru a citi informații din fișierul /etc/passwd. Această bucată de memorie a fost dezalocată de free() și apoi realocată ca buffer de citire pentru funcția exposeHeapDataTo().

În astfel de cazuri, în care se utilizează informații sensibile, este important ștergerea sau suprascrierea informațiilor sensibile înainte de a apela free(). Utilizarea funcției memset() a fost opțiunea utilizată pentru remediere. În unele cazuri, compilatoarele pot optimiza apelurile către memset() pentru buffer-ele care sunt pe cale să fie eliberate, presupunând că nu este necesară setarea cu zero a memoriei care nu mai este utilizată. Pentru a evita această posibilitate, poate fi utilizată funcția C23 memset_explicit().

Spre deosebire de memset(), funcția memset_explicit() presupune că memoria setată poate fi accesată în viitor și, prin urmare, apelul funcției nu poate fi optimizat. Dacă standardul C23 nu poate fi utilizat, funcția secure_memset() este a doua cea mai bună opțiune.

```c
// use secure_memset or C23 memset_explicit
// the compiler can optimize away the memset call
void *secure_memset(void *s, int c, size_t n)
{
    volatile unsigned char *p = s;
    while (n--)
        *p++ = c;
    return s;
}

void readSensitiveDataFrom(const char *filename)
{
    FILE *file = fopen(filename, "rb");
    if (file == NULL)
    {
        perror("Error opening file");
        return;
    }

    setvbuf(file, NULL, _IONBF, 0);

    fseek(file, 0, SEEK_END);
    long length = ftell(file);
    rewind(file);

    char *buffer = malloc(length);
    if (buffer == NULL)
    {
        perror("Memory allocation failed");
        fclose(file);
        return;
    }

    if (fread(buffer, 1, length, file) != length)
    {
        perror("Error reading file");
        fclose(file);
        memset(buffer, '\0', length);
        free(buffer);
        return;
    }

    memset(buffer, '\0', length);
    free(buffer);
    fclose(file);
}
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities$ gcc -ggdb -o Read
ingUninitializedMemoryAfter ReadingUninitializedMemoryAfter.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities$ ./ReadingUninitia
lizedMemoryAfter
The memory was deallocated but the sesitive data still resides in the heap mwahahaha
Whoops! Some data got leaked...
```



Buffer-ul intern utilizat de fread() și alte funcții I/O de gestionare a fișierelor poate fi o sursă de probleme legate de memorie sau comportament neașteptat. Biblioteca standard C utilizează de obicei un buffer pentru operațiile intrare/ieșire pentru eficiență, ceea ce înseamnă că atunci când sunt citite sau scrise într-un fișier, datele pot trece mai întâi printr-un buffer intern. Acest buffer este gestionat de bibliotecă și este separat de orice buffere alocate în mod explicit în cod. Dacă acest buffer intern afectează programul (de exemplu, prin păstrarea în memorie a datelor pentru o perioadă extinsă de timp), există posibilitatea de control sau dezactivare utilizând funcția setvbuf().

Dezactivarea buffer-ului pentru un anumit fișier înseamnă că fiecare citire sau scriere va accesa direct discul, ceea ce poate fi mai lent, dar oferă mai mult control asupra datelor. Cu citirile directe din fișier, datele merg direct în memoria dată ca parametru către fread(), fără a fi copiate în și dintr-un buffer intern. Acest lucru ar putea reduce șansa ca datele sensibile să rămână în memorie care nu este controlată direct. Înainte de utilizarea funcției setvbuf(), chiar dacă buffer-ul în care au fost citite datele sensibile a fost zeroizat înainte de eliberarea spațiului ocupat, acestea au fost afișate în urma apelului funcției exposeHeapDataTo().

În timp ce dezactivarea buffer-ului intern poate ajuta cu anumite probleme de gestionare a memoriei, aceasta poate avea un impact semnificativ asupra performanței operațiilor de intrare/ieșire. Deși ar putea părea un pas suplimentar la prima vedere, joacă de fapt un rol semnificativ în îmbunătățirea eficienței acestor operațiuni prin reducerea numărului de apeluri de sistem și, în consecință, a accesului la disc.

Fără buffer-ul intern, fiecare apel fread() sau fwrite() se traduce direct într-un apel de sistem pentru a citi sau a scrie pe disc. Apelurile de sistem sunt scumpe din punct de vedere al performanței, deoarece implică trecerea de la modul utilizator la modul kernel. Cu buffer, mai multe apeluri fread() sau fwrite() sunt acumulate. Numai atunci când acest buffer este plin sau gol rezultă un apel de sistem. De asemenea, accesarea discului este mult mai lentă decât accesarea memoriei RAM. De fiecare dată când datele sunt citite sau scrise pe un disc, există o întârziere vizibilă. Prin punerea datelor într-un buffer, numărul de ori în care programul trebuie să acceseze discul este minimizat. Datele sunt citite sau scrise în bucăți mai mari și mai contigue, ceea ce este mult mai eficient decât accesul frecvent la disc pentru cantități mici de date.

ABSENȚA VERIFICĂRII REZULTATELOR FUNCȚIILOR DE ALOCARE A MEMORIEI

O alocare eșuată apare de obicei atunci când un program încearcă să aloce un bloc foarte mare de memorie care depășește memoria disponibilă a sistemului. Când alocarea nu reușește, funcțiile de alocare a memoriei (malloc(), calloc() etc.) returnează un pointer nul. Dacă programul încearcă apoi să utilizeze acest pointer nul fără a-l verifica mai întâi, rezultă o eroare de dereferențiere a unui pointer nul.

În acest exemplu, programul încearcă să aloce un număr extrem de mare de întregi. Numărul de elemente solicitat este (size_t)-1, cea mai mare valoare pentru un întreg fără semn. Acest lucru este foarte probabil să depășească memoria disponibilă a sistemului, cauzând malloc() să eșueze și să returneze un pointer nul. Cu toate acestea, programul nu verifică dacă largeArray este nul înainte de a încerca să-l utilizeze. Acest lucru are ca rezultat o dereferențiere a unui pointer nul, o încercare de citire sau scriere într-o zonă de memorie care nu este mapată, declanșând o eroare de tip segmentation fault sau access violation.

C LargeAllocationBefore.c > ...

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

int main() {
    size_t largeSize = (size_t)-1;
    int *largeArray = malloc(largeSize * sizeof(int));

    largeArray[0] = 10;
    printf("First element: %d\n", largeArray[0]);

    free(largeArray);
    return 0;
}
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingRetu
rnValues$ gcc -o LargeAllocationBefore LargeAllocationBefore.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingRetu
rnValues$ ./LargeAllocationBefore
Segmentation fault (core dumped)
```

O bună practică este verificarea valorilor returnate de funcțiile de alocare a memoriei și gestionarea valorilor nule. În această versiune revizuită, programul verifică dacă largeArray este nul înainte de a-l utiliza, evitând astfel, o potențială eroare de dereferențiere a unui pointer nul. Gestionarea robustă a erorilor și gestionarea resurselor sensibile sunt esențiale pentru scrierea unui software stabil și sigur.

LargeAllocationAfter.c > ...

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

int main() {
    size_t largeSize = (size_t)-1;
    int *largeArray = malloc(largeSize * sizeof(int));

    if (largeArray == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    largeArray[0] = 10;
    printf("First element: %d\n", largeArray[0]);

    free(largeArray);
    return 0;
}
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingRetu
rnValues$ gcc -o LargeAllocationAfter LargeAllocationAfter.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingRetu
rnValues$ ./LargeAllocationAfter
Memory allocation failed
```

Memoria este o resursă limitată și poate fi epuizată. Memoria disponibilă este de obicei limitată de suma cantității de memorie fizică și a spațiului swap alocat sistemului de operare. Un sistem poate fi capabil să aloce cel mult atât spațiu heap tuturor proceselor care rulează (minus dimensiunea sistemului de operare în sine și segmentelor de text și date ale tuturor proceselor care rulează). Odată ce toată memoria virtuală este alocată, solicitările pentru mai multă memorie vor eșua.

Epuizarea spațiului din memoria heap poate rezulta dintr-o serie de cauze, inclusiv leak-uri de memorie (memoria alocată dinamic care nu este eliberată după ce nu mai este necesară) sau din cauza altor procese.

ABSENȚA VERIFICĂRII DIMENSIUNII MEMORIEI CE VA FI ALOCATĂ

O problemă serioasă poate fi cauzată atunci când se încearcă alocarea de memorie folosind malloc() cu o dimensiune care se bazează pe o variabilă și acea variabilă ajunge să fie un număr negativ.

În exemplu, dacă variabila size este -1, malloc() este apelat cu o dimensiune de 0, care este legală în C, dar are ca rezultat fie un pointer nul, fie un pointer unic care poate fi dat ca parametru în siguranță funcției free(), dar care nu poate fi dereferențiat.

Când malloc(0) returnează un pointer non-NULL, returnează un pointer unic ca valoare. Acest pointer este o adresă validă în spațiul de adrese al programului, dar nu indică un bloc de memorie alocată pe care poate fi utilizat. Locația memoriei spre care indică indicatorul nu este destinată stocării datelor. Aceast lucru înseamnă că nu ar trebui citite sau scrise date în această locație, o astfel de încercare putând duce la un comportament nedefinit. Memoria în care sunt stocate șirurile literale este de obicei marcată ca fiind read-only. Încercarea de a modifica un șir literal are ca rezultat un comportament nedefinit, ceea ce duce adesea la o eroare de execuție, cum ar fi o eroare de tip segmentation fault. Motivul pentru returnarea unui pointer unic, non-NULL (în loc să fie returnat doar NULL) este de a face diferența între o cerere de alocare reușită de zero octeți și o alocare nereușită (nu mai este memorie).

```
5   char *generateRandomCharArray(int size)
6   {
7       char *array = (char *)malloc((size + 1) * sizeof(char));
8       if (array == NULL)
9       {
10          perror("Memory allocation failed");
11          return NULL;
12      }
13
14      srand(time(NULL));
15
16      for (int i = 0; i < size; i++)
17      {
18          array[i] = 'a' + (rand() % 26);
19      }
20
21      array[size]='\0';
22
23      return array;
24  }
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingReturn
Values$ gcc -ggdb -o ZeroSizeAllocationBefore  ZeroSizeAllocationBefore.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingReturn
Values$ ./ZeroSizeAllocationBefore
Enter the size of the random string: -1
```

Comanda x/10x array examinează 10 cuvinte de memorie în format hexazecimal pornind de la adresa indicată de array. Rezultatul arată valorile hexazecimale stocate în acea locație (0x65746e45/0x68742072/0x69732065/0x6f20657a/0x68742066 etc.). Analizând aceste date (etnE/ht

r/is e/o ez/ht f), se pare că aceasta este memoria read-only în care sunt stocate șirurile literale (Ente/r th/e si/ze o/f th).

```
(gdb) b 26
Breakpoint 1 at 0x12ed: file ZeroSizeAllocationBefore.c, line 27.
(gdb) r
Starting program: /home/denissa/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingReturnValues/ZeroSizeAllocationBefore

(gdb) x/10x array
0x5555555592a0: 0x65746e45      0x68742072      0x69732065      0x6f20657a
0x5555555592b0: 0x68742066      0x61722065      0x6d6f646e      0x72747320
0x5555555592c0: 0x3a676e69      0x00000320
(gdb) n
28          return array;
(gdb) x/10x array
0x555555559ac0: 0x000005f0      0x000005f1      0x000005f2      0x000005f3
0x555555559ad0: 0x000005f4      0x000005f5      0x00020531      0x00000000
0x555555559ae0: 0x000005f8      0x000005f9
```

Accesarea unui index în afara limitelor tabloului nu termină întotdeauna abrupt execuția programului. Dacă memoria accesată de -1 este sub controlul programului, atunci va apărea o valoare nedefinită (care a fost creată de program anterior). Conform standardului, accesarea unui element din afara limitelor tabloului invocă un comportament nedefinit. Schimbarea adresei de memorie observată în GDB după executarea array[-1]='\0' sugerează un comportament nedefinit în program. Scrierea în array[-1] reprezintă accesarea memoriei înainte de începerea blocului alocat, ceea ce poate corupe memoria și poate duce la un comportament imprevizibil.

Pentru a evita astfel de erori, o bună practică recomandată este asigurarea faptului că valorile utilizate pentru numărul de elemente ce va fi alocat se încadrează în intervale valide și nu sunt negative.

În versiunea corectată a codului alocarea memoriei are loc numai dacă numărul de elemente nu este negativ, asigurându-se că dimensiunea pentru malloc() este validă.

```c
char *generateRandomCharArray(int size)
{
    if (size <= 0)
    {
        return NULL;
    }

    char *array = (char *)malloc((size + 1) * sizeof(char));
    if (array == NULL)
    {
        perror("Memory allocation failed");
        return NULL;
    }

    srand(time(NULL));

    for (int i = 0; i < size; i++)
    {
        array[i] = 'a' + (rand() % 26);
    }

    array[size]='\0';

    return array;
}
```

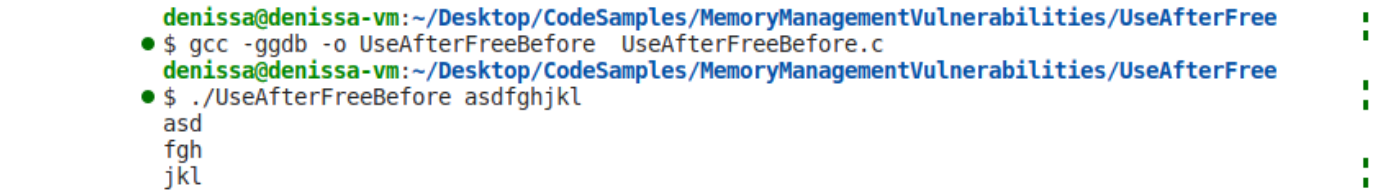
```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingReturn
Values$ gcc -ggdb -o ZeroSizeAllocationAfter ZeroSizeAllocationAft
er.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/NotCheckingReturn
Values$ ./ZeroSizeAllocationAfter
Enter the size of the random string: -1
Failed to allocate memory or invalid size.
```

REFERENȚIEREA MEMORIEI DEZALOCATE

În aplicațiile mari sau complexe în care memoria este frecvent alocată și dezalocată, mai ales dacă există gestionare manuală a memoriei (folosind malloc(), free(), etc.), urmărirea stării actuale a fiecărei alocări poate deveni o provocare. Gestionarea memoriei poate fi, de asemenea, o victimă a erorilor tipice date de copy-paste.

Referențierea memoriei eliberate, cunoscută și ca vulnerabilitatea "use-after-free", apare atunci când un program continuă să utilizeze un pointer către o zonă de memorie după ce aceasta a fost eliberată. Acest lucru poate duce la un comportament imprevizibil, inclusiv terminări abrupte ale programului, coruperea datelor sau vulnerabilități de securitate. Acesta este un exemplu ce ilustrează cum ar putea arăta un scenariu "use-after-free" și cum poate fi evitat.

```
11      int size = strlen(argv[1]) / 3;
12      char *buf1R1 = (char *)malloc(size + 1);
13      if (buf1R1 == NULL) {
14          perror("Failed to allocate memory for buf1R1");
15          return 1;
16      }
17      strncpy(buf1R1, argv[1], size);
18      buf1R1[size] = '\0';
19      printf("%s\n", buf1R1);
20      free(buf1R1);
21
22      char *buf2R1 = (char *)malloc(size + 1);
23      if (buf2R1 == NULL) {
24          perror("Failed to allocate memory for buf2R1");
25          return 1;
26      }
27      strncpy(buf2R1, argv[1] + size, size);
28      buf2R1[size] = '\0';
29      printf("%s\n", buf2R1);
30      free(buf2R1);
31
32      char *buf2R2 = (char *)malloc(size + 1);
33      if (buf2R2 == NULL) {
34          perror("Failed to allocate memory for buf2R2");
35          return 1;
36      }
37      strncpy(buf2R1, argv[1] + 2 * size, size);
38      buf2R2[size] = '\0';
39      printf("%s\n", buf2R2);
40      free(buf2R2);
```



Variabila size este calculată ca o treime din lungimea argv[1]. Apoi, pentru fiecare dintre cele trei părți, codul alocă dinamic memorie pentru un buffer (buf1R1, buf2R1 și buf2R2) pentru a ține fiecare parte, cu un caracter suplimentar pentru terminatorul nul. După copierea subșirului relevant în fiecare buffer, codul îl afișează și apoi eliberează memoria alocată. Problema din acest cod este buffer-ul buf2R1 utilizat în al treilea bloc de instrucțiuni (ar trebui să fie buf2R2). După eliberarea memoriei pentru buf2R1, programul încearcă să copieze a treia parte a șirului în acea locație. Deoarece memoria a fost deja eliberată, această operație este nesigură și poate duce la un comportament nedefinit.

Pentru a evita vulnerabilitățile de utilizare după dealocare, o bună practică este setarea pointer-ilor cu valoarea NULL după eliberare. Acest lucru împiedică utilizarea memoriei eliberate, deoarece orice încercare de a dereferenția un pointer NULL va fi mai previzibilă (cauzând de obicei o terminare bruscă a programului). Altfel, programul se poate termina abrupt, poate produce rezultate incorecte sau chiar executa cod rău intenționat în contextul unei aplicații mai complexe. În acest caz rezultatul este cel dorit, dar în mod eronat, făcând greșeala și mai greu de identificat.

```c
    int size = strlen(argv[1]) / 3;
    char *buf1R1 = (char *)malloc(size + 1);
    if (buf1R1 == NULL)
    {
        perror("Failed to allocate memory for buf1R1");
        return 1;
    }
    strncpy(buf1R1, argv[1], size);
    buf1R1[size] = '\0';
    printf("%s\n", buf1R1);
    free(buf1R1);
    buf1R1 = NULL;

    char *buf2R1 = (char *)malloc(size + 1);
    if (buf2R1 == NULL)
    {
        perror("Failed to allocate memory for buf2R1");
        return 1;
    }
    strncpy(buf2R1, argv[1] + size, size);
    buf2R1[size] = '\0';
    printf("%s\n", buf2R1);
    free(buf2R1);
    buf2R1 = NULL;

    char *buf2R2 = (char *)malloc(size + 1);
    if (buf2R2 == NULL)
    {
        perror("Failed to allocate memory for buf2R2");
        return 1;
    }
    strncpy(buf2R1, argv[1] + 2 * size, size);
    buf2R2[size] = '\0';
    printf("%s\n", buf2R2);
    free(buf2R2);
    buf2R2 = NULL;
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/UseAfterFree
$ gcc -ggdb -o UseAfterFreeAfter  UseAfterFreeAfter.c
denissa@denissa-vm:~/Desktop/CodeSamples/MemoryManagementVulnerabilities/UseAfterFree
$ ./UseAfterFreeAfter asdfghjkl
asd
fgh
Segmentation fault (core dumped)
```

În versiunea corectată, după eliberarea buffere-lor, acestea sunt setate cu NULL. Acest lucru face programul mai sigur, deoarece orice utilizare ulterioară a oricărui buffer va fi mai ușor de detectat și de gestionat.