**Управление памятью**

**MEM01-C. Храните новое значение в указателе сразу после вызова функции free()**

Висячие указатели могут привести к тому, что дважды будет вызвана функции free(). Простой, но эффективный способ избавления от висячих указателей и предотвращения многих уязвимостей, связанных с памятью, это установить указатели в NULL после того как они будут освобождены или установить их на другой действительный объект.

В этом примере не компилируемого кода, message\_type используется, чтобы определить, как обрабатывать само сообщение. Предполагается, что message\_type является целым числом и сообщение является указателем на массив символов, которые были выделены динамически. Если message\_type=value\_1, сообщение обрабатывается соответствующим образом. Аналогичная операция происходит, когда message\_type= value \_2. Однако, если message\_type == value \_1 истиннно и message\_type == value \_2 также истинно, то сообщение освобождается дважды.

char \*message;

int message\_type;

/\* Initialize message and message\_type \*/

if (message\_type == value\_1) {

  /\* Process message type 1 \*/

  free(message);

}

/\* ...\*/

if (message\_type == value\_2) {

   /\* Process message type 2 \*/

  free(message);

}

Установка message в NULL после того, как оно освобождается, исключает возможность того, что message\_type может быть использовано при освобождении памяти более одного раза.

char \*message;

int message\_type;

/\* Initialize message and message\_type \*/

if (message\_type == value\_1) {

  /\* Process message type 1 \*/

  free(message);

  message = NULL;

}

/\* ... \*/

if (message\_type == value\_2) {

  /\* Process message type 2 \*/

  free(message);

  message = NULL;}

**MEM30-C. Не обращайтесь к освобожденной памяти**

Поведение указателей в памяти, которые были освобождены функциями free() или realloc(), не определено. Указатели на память, которая была освобождена, называются висячими ссылками. Доступ к висячим ссылкам может привести к уязвимостям.

Чтение указателя освобожденной памяти не определено, потому что значение указателя является неопределенным. Когда память освобождается, все указатели в ней становятся недействительными, и их содержание может быть либо возвращено операционной системе, что делает освободившееся пространство недоступным, либо остаются нетронутыми и доступны. В результате, данные в освободившемся месте могут оказаться действительным, но неожиданно измениться. Следовательно, память не должны быть записана или прочитана, когда она будет освобождена.

В примере р освобождается до р->next . Следовательно р->next считывает память, которая уже была освобождена.

#include <stdlib.h>

struct node {

  int value;

  struct node \*next;

};

void free\_list(struct node \*head) {

  for (struct node \*p = head; p != NULL; p = p->next) {

    free(p);

  }

}

Исправим ошибку, сохраняя ссылку на р->next в q до освобождения р:

#include <stdlib.h>

struct node {

  int value;

  struct node \*next;

};

void free\_list(struct node \*head) {

  struct node \*q;

  for (struct node \*p = head; p != NULL; p = q) {

    q = p->next;

    free(p);

  }

}

В этом примере не компилируемого кода, BUF записывается после того, как был освобожден. Запись после освобождения может привести к запуску произвольного кода.

#include <stdlib.h>

#include <string.h>

int main(int argc, char \*argv[]) {

  char \*return\_val = 0;

  const size\_t bufsize = strlen(argv[0]) + 1;

  char \*buf = (char \*)malloc(bufsize);

  if (!buf) {

    return EXIT\_FAILURE;

  }

  /\* ... \*/

  free(buf);

  /\* ... \*/

  strcpy(buf, argv[0]);

  /\* ... \*/

  return EXIT\_SUCCESS;

}

В этом компилируемом коде память освобождается после конечного использования

#include <stdlib.h>

#include <string.h>

int main(int argc, char \*argv[]) {

  char \*return\_val = 0;

  const size\_t bufsize = strlen(argv[0]) + 1;

  char \*buf = (char \*)malloc(bufsize);

  if (!buf) {

    return EXIT\_FAILURE;

  }

  /\* ... \*/

  strcpy(buf, argv[0]);

  /\* ... \*/

  free(buf);

  return EXIT\_SUCCESS;

}

В этом не компилируемом коде возвращаемое значение gdRealloc (простая функция ReAlloc (), которая перераспределяет пространство, на которое указывает im->clip->list) равно more. Тем не менее, значение im->clip->list используется непосредственно после этого в коде, и C стандарт устанавливает, что если Realloc () перемещает указанную область, то исходный блок освобождается.

void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {

  gdClipRectanglePtr more;

  if (im->clip == 0) {

   /\* ... \*/

  }

  if (im->clip->count == im->clip->max) {

    more = gdRealloc (im->clip->list,(im->clip->max + 8) \*

                      sizeof (gdClipRectangle));

    /\*

     \* If the realloc fails, then we have not lost the

     \* im->clip->list value.

     \*/

    if (more == 0) return;

    im->clip->max += 8;

  }

  im->clip->list[im->clip->count] = \*rect;

  im->clip->count++;

Решение проблемы в присваивании im->clip->list к значению more после вызова ReAlloc ().

void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {

  gdClipRectanglePtr more;

  if (im->clip == 0) {

    /\* ... \*/

  }

  if (im->clip->count == im->clip->max) {

    more = gdRealloc (im->clip->list,(im->clip->max + 8) \*

                      sizeof (gdClipRectangle));

    if (more == 0) return;

    im->clip->max += 8;

    im->clip->list = more;

  }

  im->clip->list[im->clip->count] = \*rect;

  im->clip->count++;

**MEM34-C. Только освобожденная память выделяется динамически**

Освобождение памяти, которая не выделяется динамически, может привести к серьезным ошибкам. Не вызывайте free () на указатели, кроме тех, которые возвращаются стандартными функциями выделения памяти, таких как malloc (), calloc (), Realloc (), или aligned\_alloc ().

Аналогичная ситуация возникает, когда аргументом Realloc () является указатель на нединамически выделенную память. Функция Realloc () используется для изменения размеров блока динамической памяти. Если аргументом Realloc () является указатель на область памяти, не распределенную по стандартной функции выделения памяти, поведение не определено.

Это правило не распространяется на нулевые указатели. C Стандарт гарантирует, что если в free() передается пустой указатель, не происходит никаких действий.

Этот неправильный пример кода, c\_str ссылаться либо на динамически выделенную память, либо на статически выделяемую строку в зависимости от значения arg. В любом случае, c\_str передается в качестве аргумента, в функцию free ().

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

enum { MAX\_ALLOCATION = 1000 };

int main(int argc, const char \*argv[]) {

  char \*c\_str = NULL;

  size\_t len;

  if (argc == 2) {

    len = strlen(argv[1]) + 1;

    if (len > MAX\_ALLOCATION) {

      /\* Handle error \*/

    }

    c\_str = (char \*)malloc(len);

    if (c\_str == NULL) {

      /\* Handle error \*/

    }

    strcpy(c\_str, argv[1]);

  } else {

    c\_str = "usage: $>a.exe [string]";

    printf("%s\n", c\_str);

  }

  free(c\_str);

  return 0;

}

Это решение исключает возможность c\_str ссылаться к памяти, которая не размещена динамически.

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

enum { MAX\_ALLOCATION = 1000 };

int main(int argc, const char \*argv[]) {

  char \*c\_str = NULL;

  size\_t len;

  if (argc == 2) {

    len = strlen(argv[1]) + 1;

    if (len > MAX\_ALLOCATION) {

      /\* Handle error \*/

    }

    c\_str = (char \*)malloc(len);

    if (c\_str == NULL) {

      /\* Handle error \*/

    }

    strcpy(c\_str, argv[1]);

  } else {

    printf("%s\n", "usage: $>a.exe [string]");

    return EXIT\_FAILURE;

  }

  free(c\_str);

  return 0;

}

В этом неправильном примере, параметр указатель в realloc (),buf, не относится к динамически выделяемой памяти:

#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {

  char buf[BUFSIZE];

  char \*p = (char \*)realloc(buf, 2 \* BUFSIZE);

  if (p == NULL) {

    /\* Handle error \*/

  }

}

Решение в том, чтобы выделить buf динамически

#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {

  char \*buf = (char \*)malloc(BUFSIZE \* sizeof(char));

  char \*p = (char \*)realloc(buf, 2 \* BUFSIZE);

  if (p == NULL) {

    /\* Handle error \*/

  }

}