Karthikeyan G
CB.SC.P2CYS24008

Secure Coding Lab - 11
Rule 08. Memory Management

MEM30-C. Do not access freed memory:

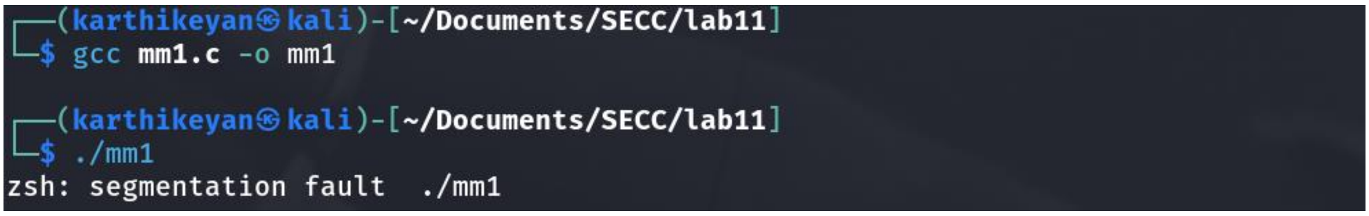
- Evaluating a pointer to deallocated memory, including dereferencing, arithmetic operations, type casting, or assignment, results in undefined behavior.
- Deallocated memory pointers are known as dangling pointers, and accessing them can lead to exploitable vulnerabilities.
- The C Standard states that using a pointer value from memory freed by free() or realloc() is undefined behavior.
- Reading from a pointer to deallocated memory is undefined because the pointer value is indeterminate and may be a trap representation.
- The memory manager decides when to reallocate or recycle freed memory; once memory is freed, all pointers to it become invalid.
- Freed memory contents may be returned to the operating system or remain intact but can change unexpectedly.
- Memory should not be read from or written to after it has been freed.

Noncompliant Code Example

This example shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, p is freed before p->next is executed, so that p->next reads memory that has already been freed.

```
#include <stdlib.h>
#include <stdio.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);
 }
}
```

```
int main() {
 struct node *head = (struct node *)malloc(sizeof(struct node));
 head->value = 1;
 head->next = (struct node *)malloc(sizeof(struct node));
 head->next->value = 2;
 head->next->next = (struct node *)malloc(sizeof(struct node));
 head->next->next->value = 3;
 head->next->next->next = NULL;
 free_list(head);
 printf("First value after free: %d\n", head->value); // ERROR
 return 0;
}
```

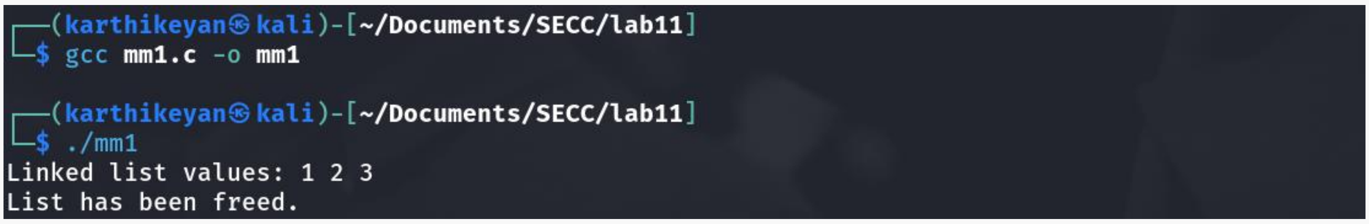


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm1.c -o mm1

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm1
zsh: segmentation fault  ./mm1
```

Compliant Solution

Correct this error by storing a reference to p->next in q before freeing p:

```
#include <stdio.h>
#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);
 }
}
int main() {
 struct node *head = NULL;
 struct node *second = NULL;
 struct node *third = NULL;
 head = (struct node*)malloc(sizeof(struct node));
 second = (struct node*)malloc(sizeof(struct node));
 third = (struct node*)malloc(sizeof(struct node));
 head->value = 1;
 head->next = second;
 second->value = 2;
 second->next = third;
 third->value = 3;
 third->next = NULL;
 printf("Linked list values: ");
 for (struct node *p = head; p != NULL; p = p->next) {
 printf("%d ", p->value);
 }
 printf("\n");
 free_list(head);
 head = NULL;
 printf("List has been freed.\n");
 return 0;
}
```

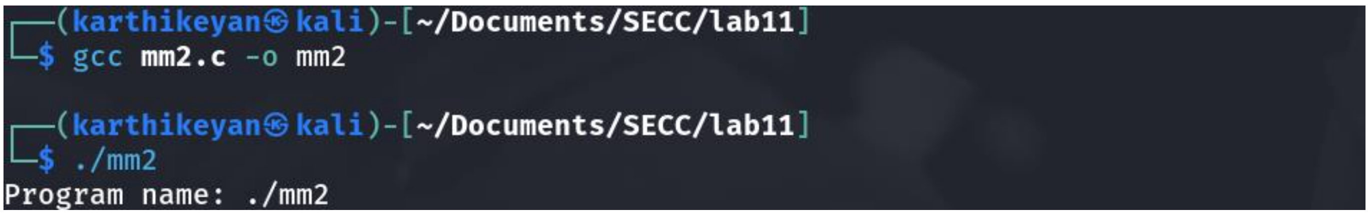


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm1.c -o mm1

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm1
Linked list values: 1 2 3
List has been freed.
```

Noncompliant Code Example

In this noncompliant code example, buf is written to after it has been freed. Write-after-free vulnerabilities can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, allocations and frees are far removed, making it difficult to recognize and diagnose these problems.

```
#include <stdio.h>
#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) {
 printf("Memory allocation failed.\n");
 return EXIT_FAILURE;
 }
 strcpy(buf, argv[0]);
 printf("Program name: %s\n", buf);
 free(buf);
 return EXIT_SUCCESS;
}
```

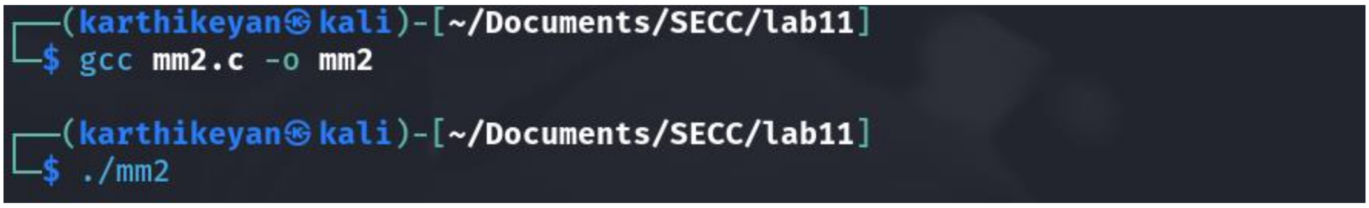


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm2.c -o mm2

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm2
Program name: ./mm2
```

Compliant Solution

In this compliant solution, the memory is freed after its final use:

```
#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;
}
```

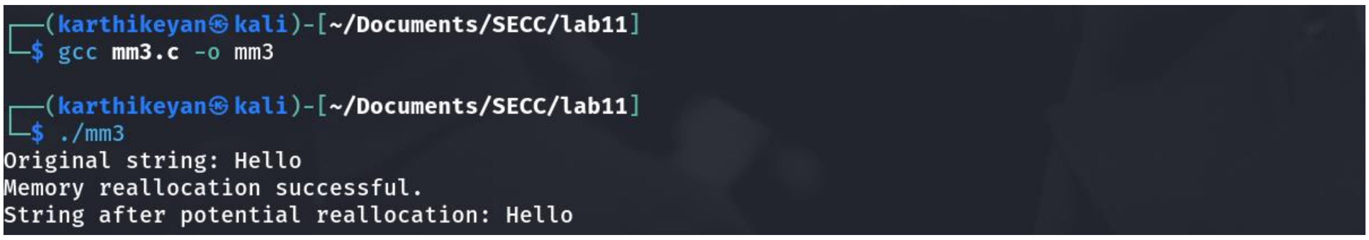

```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm2.c -o mm2

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm2
```

Noncompliant Code Example

In this noncompliant example, realloc() may free c_str1 when it returns a null pointer, resulting in c_str1 being freed twice. The C Standards Committee's proposed response to Defect Report #400 makes it implementation-defined whether or not the old object is deallocated when size is zero and memory for the new object is not allocated. The current implementation of realloc() in the GNU C Library and Microsoft Visual Studio's Runtime Library will free c_str1 and return a null pointer for zero byte allocations. Freeing a pointer twice can result in a potentially exploitable vulnerability commonly referred to as a double-free vulnerability

```
#include <stdio.h>
#include <stdlib.h>
void f(char *c_str1, size_t size) {
 char *c_str2 = (char *)realloc(c_str1, size);
 if (c_str2 == NULL) {
 free(c_str1);
 printf("Memory reallocation failed and original memory freed.\n");
 } else {
```

```
 printf("Memory reallocation successful.\n");
}
}
int main() {
 size_t initial_size = 10;
 char *my_string = (char *)malloc(initial_size);
 if (my_string == NULL) {
 printf("Initial memory allocation failed.\n");
 return EXIT_FAILURE;
 }
 snprintf(my_string, initial_size, "Hello");
 printf("Original string: %s\n", my_string);
 f(my_string, 20);
 if (my_string) {
 printf("String after potential reallocation: %s\n", my_string);
 }
 free(my_string);
 return EXIT_SUCCESS;
}
```

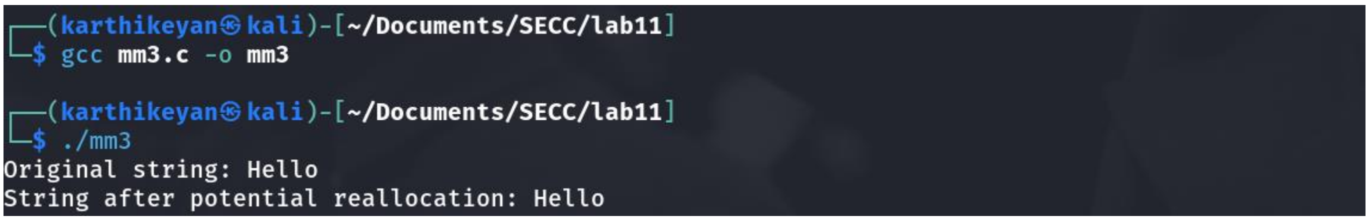


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm3.c -o mm3

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm3
Original string: Hello
Memory reallocation successful.
String after potential reallocation: Hello
```

Compliant Solution

This compliant solution does not pass a size argument of zero to the realloc() function, eliminating the possibility of c_str1 being freed twice:

```
#include <stdio.h>
#include <stdlib.h>
void f(char *c_str1, size_t size) {
if (size != 0) {
 char *c_str2 = (char *)realloc(c_str1, size);
 if (c_str2 == NULL) {
 free(c_str1);
 }
 } else {
 free(c_str1);
 }
}
int main() {
 size_t initial_size = 10;
 char *my_string = (char *)malloc(initial_size);
 if (my_string == NULL) {
 return EXIT_FAILURE;
 }
 snprintf(my_string, initial_size, "Hello");
 printf("Original string: %s\n", my_string);
 f(my_string, 20);
 printf("String after potential reallocation: %s\n", my_string);
 free(my_string);
 return EXIT_SUCCESS;
}
```



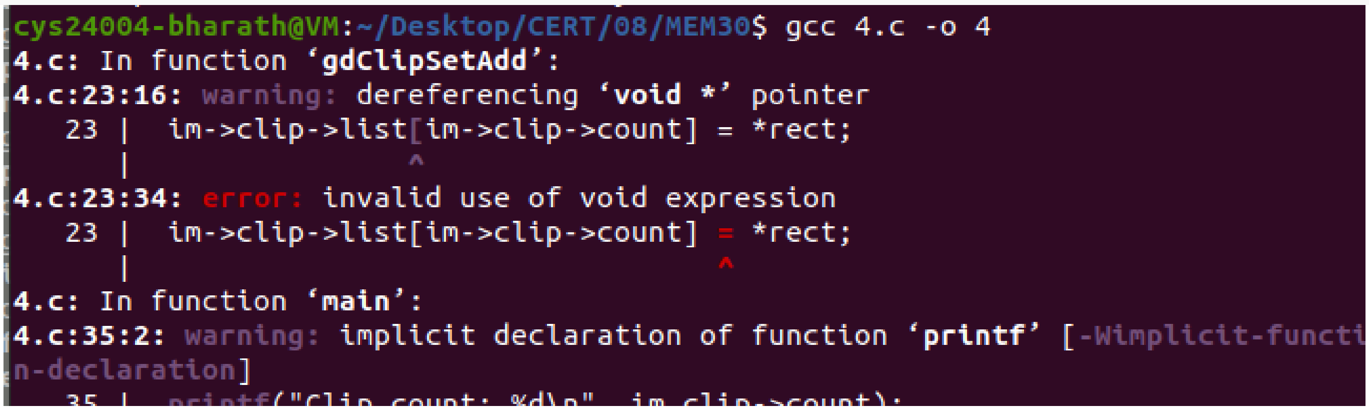
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm3.c -o mm3

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm3
Original string: Hello
String after potential reallocation: Hello
```

If the intent of calling f() is to reduce the size of the object, then doing nothing when the size is zero would be unexpected; instead, this compliant solution frees the object.

Noncompliant Code Example

In this noncompliant example (CVE-2009-1364) from libwmf version 0.2.8.4, the return value of gdRealloc (a simple wrapper around realloc() that reallocates space pointed to by im->clip->list) is set to more. However, the value of im->clip->list is used directly afterwards in the code, and the C Standard specifies that if realloc() moves the area pointed to, then the original block is freed. An attacker can then execute arbitrary code by forcing a reallocation (with a sufficient im->clip->count) and accessing freed memory

```
#include <stdlib.h>
typedef struct {
 int count;
 int max;
 void *list;
} gdClip;
typedef struct {
 int x, y, width, height;
} gdClipRectangle;
typedef struct {
 gdClip *clip;
} gdImage;
void gdClipSetAdd(gdImage *im, gdClipRectangle *rect) {
 gdClipRectangle *more;
 if (im->clip == 0) {
 return;
 }
 if (im->clip->count == im->clip->max) {
 more = realloc(im->clip->list, (im->clip->max + 8) * sizeof(gdClipRectangle));
 if (more == 0) return;
 im->clip->max += 8;
 }
 im->clip->list[im->clip->count] = *rect;
 im->clip->count++;
}
int main() {
 gdImage im;
 gdClip clip;
 gdClipRectangle rect = {0, 0, 100, 100};
 clip.count = 0;

 clip.max = 8;

 clip.list = malloc(clip.max * sizeof(gdClipRectangle));

 im.clip = &clip;

 gdClipSetAdd(&im, &rect);

 printf("Clip count: %d\n", im.clip->count);

 free(im.clip->list);

 return EXIT_SUCCESS;

}
```

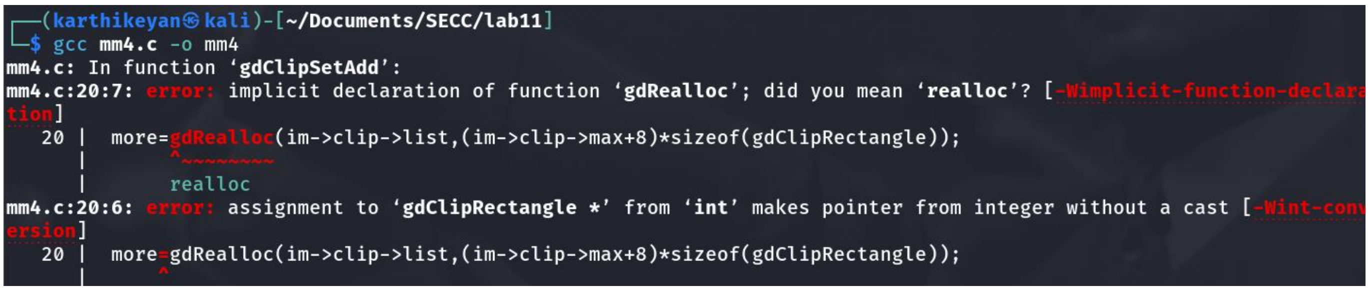


```
cys24004-bharath@VM:~/Desktop/CERT/08/MEM30$ gcc 4.c -o 4
4.c: In function 'gdClipSetAdd':
4.c:23:16: warning: dereferencing 'void *' pointer
   23 |   im->clip->list[im->clip->count] = *rect;
      |                 ^
4.c:23:34: error: invalid use of void expression
   23 |   im->clip->list[im->clip->count] = *rect;
      |                                   ^
4.c: In function 'main':
4.c:35:2: warning: implicit declaration of function 'printf' [-Wimplicit-functi
n-declaration]
   35 |   printf("Clip count: %d\n", im.clip->count);
```

Compliant Solution

This compliant solution simply reassigns im->clip->list to the value of more after the call to realloc():

```
#include <stdio.h>

#include <stdlib.h>

typedef struct {

 int count;

 int max;

 void *list;

} gdClip;

typedef struct {

 int x,y,width,height;

} gdClipRectangle;

typedef struct {
 gdClip *clip;
} gdImage;
void gdClipSetAdd(gdImage *im, gdClipRectangle *rect) {
 gdClipRectangle *more;
 if (im->clip==0) {
 return;
 }
 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++;
}
int main() {
 gdImage im;
 gdClip clip;
 gdClipRectangle rect={0,0,100,100};
 clip.count=0;
 clip.max=8;
 clip.list=malloc(clip.max*sizeof(gdClipRectangle));
 im.clip=&clip;
gdClipSetAdd(&im,&rect);
 printf("Clip count: %d\n",im.clip->count);
 free(im.clip->list);
 return EXIT_SUCCESS;
}
```



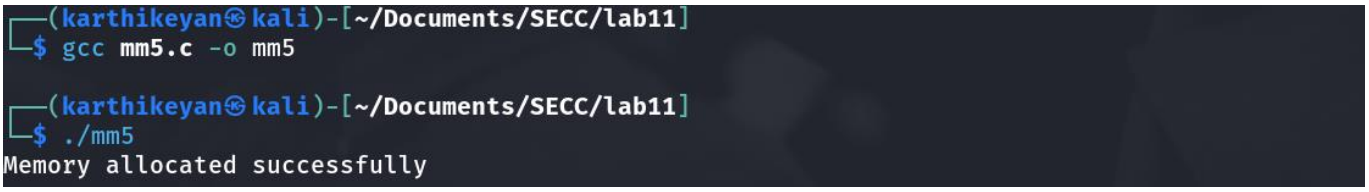
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm4.c -o mm4
mm4.c: In function ‘gdClipSetAdd’:
mm4.c:20:7: error: implicit declaration of function ‘gdRealloc’; did you mean ‘realloc’? [-Wimplicit-function-declara
tion]
   20 |  more=gdRealloc(im->clip->list,(im->clip->max+8)*sizeof(gdClipRectangle));
      |       ^~~~~~~~~
      |       realloc
mm4.c:20:6: error: assignment to ‘gdClipRectangle *’ from ‘int’ makes pointer from integer without a cast [-Wint-conv
ersion]
   20 |  more=gdRealloc(im->clip->list,(im->clip->max+8)*sizeof(gdClipRectangle));
      |      ^
```

MEM31-C. Free dynamically allocated memory when no longer needed

Noncompliant Code Example

In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text_buffer referring to the object:

```
#include <stdio.h>
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };
int f(void) {
 char *text_buffer = (char *)malloc(BUFFER_SIZE);
 if (text_buffer == NULL) {
 return -1;
 }
 return 0;
}
int main() {
 if (f() == -1) {
 printf("Memory allocation failed\n");
 return EXIT_FAILURE;
 }
 printf("Memory allocated successfully\n");
 return EXIT_SUCCESS;
}
```

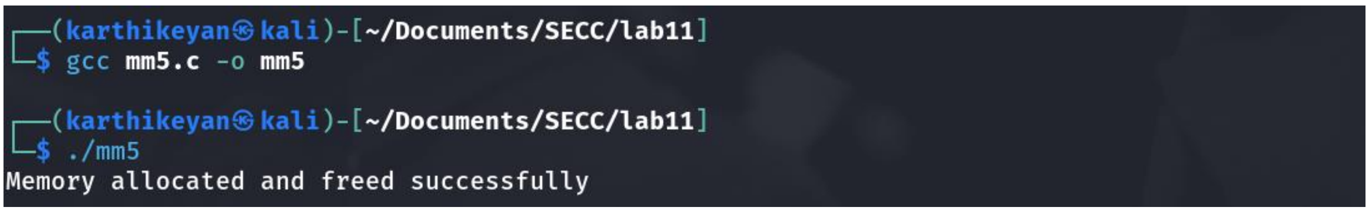


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm5.c -o mm5

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm5
Memory allocated successfully
```

Compliant Solution

In this compliant solution, the pointer is deallocated with a call to free():

```
#include <stdio.h>

#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {

 char *text_buffer = (char *)malloc(BUFFER_SIZE);

 if (text_buffer == NULL) {

 return -1;

 }

 free(text_buffer);

 return 0;

}

int main() {

 if (f() == -1) {

 printf("Memory allocation failed\n");

 return EXIT_FAILURE;

 }

 printf("Memory allocated and freed successfully\n");

 return EXIT_SUCCESS;

}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm5.c -o mm5

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm5
Memory allocated and freed successfully
```

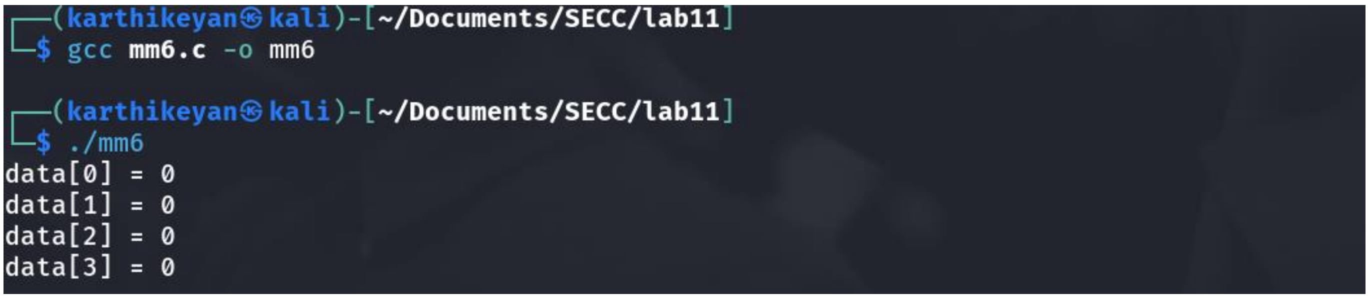
}

MEM33-C. Allocate and copy structures containing a flexible array member dynamically

Noncompliant Code Example (Storage Duration)

This noncompliant code example uses automatic storage for a structure containing a flexible array member:

```
#include <stdlib.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void func(void) {
 struct flex_array_struct *flex_struct = malloc(sizeof(struct flex_array_struct) + 4 *
sizeof(int));
 size_t array_size = 4;
 if (flex_struct == NULL) {
return;
 }
 flex_struct->num = array_size;
 for (size_t i = 0; i < array_size; ++i) {
 flex_struct->data[i] = 0;
 }
 free(flex_struct);
}
int main() {
 func();
 printf("Flex array initialized successfully\n");
 return EXIT_SUCCESS;
}
```



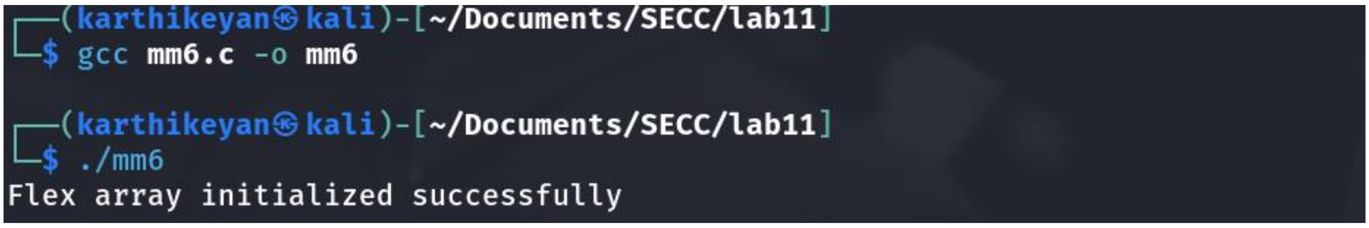
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm6.c -o mm6

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm6
data[0] = 0
data[1] = 0
data[2] = 0
data[3] = 0
```

Because the memory for flex_struct is reserved on the stack, no space is reserved for the data member. Accessing the data member is undefined behavior.

Compliant Solution (Storage Duration)

This compliant solution dynamically allocates storage for flex_array_struct:

```
#include <stdio.h>
#include <stdlib.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void func(void) {
 struct flex_array_struct *flex_struct;
 size_t array_size = 4;
 flex_struct = (struct flex_array_struct *)malloc(sizeof(struct flex_array_struct) + sizeof(int)
*
array_size);
 if (flex_struct == NULL) {
 return;
 }
 flex_struct->num = array_size;
 for (size_t i = 0; i < array_size; ++i) {
 flex_struct->data[i] = 0;
 }
 free(flex_struct);
}
int main() {
 func();
 printf("Flex array initialized successfully\n");
 return EXIT_SUCCESS;
}
```

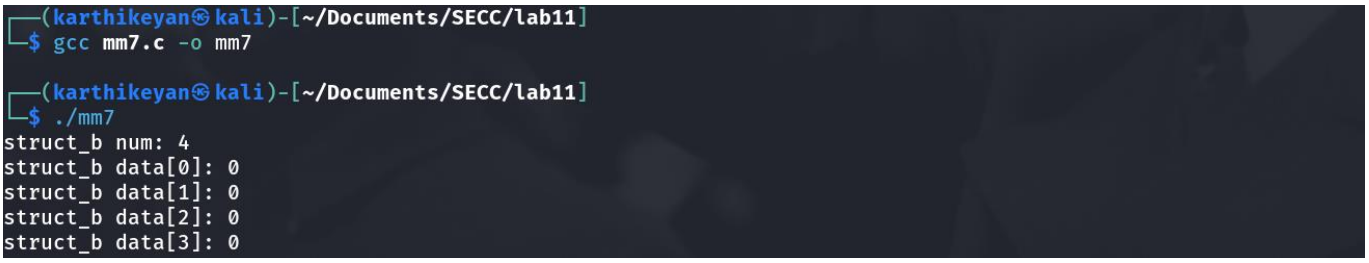


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm6.c -o mm6

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm6
Flex array initialized successfully
```

Noncompliant Code Example (Copying)

This noncompliant code example attempts to copy an instance of a structure containing a flexible array member (struct flex_array_struct) by assignment:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void func(struct flex_array_struct *struct_a, struct flex_array_struct *struct_b) {
 *struct_b = *struct_a;
}
int main() {
 size_t array_size = 4;
 struct flex_array_struct *struct_a = malloc(sizeof(struct flex_array_struct) + sizeof(int) *
array_size);
 struct flex_array_struct *struct_b = malloc(sizeof(struct flex_array_struct) + sizeof(int) *
array_size);
 if (struct_a == NULL || struct_b == NULL) {
 return EXIT_FAILURE;
 }
 struct_a->num = array_size;
 for (size_t i = 0; i < array_size; ++i) {
 struct_a->data[i] = i;
 }
 func(struct_a, struct_b);
 printf("struct_b num: %zu\n", struct_b->num);
 for (size_t i = 0; i < struct_b->num; ++i) {
 printf("struct_b data[%zu]: %d\n", i, struct_b->data[i]);
 }
 free(struct_a);
 free(struct_b);
 return EXIT_SUCCESS;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm7.c -o mm7

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm7
struct_b num: 4
struct_b data[0]: 0
struct_b data[1]: 0
struct_b data[2]: 0
struct_b data[3]: 0
```

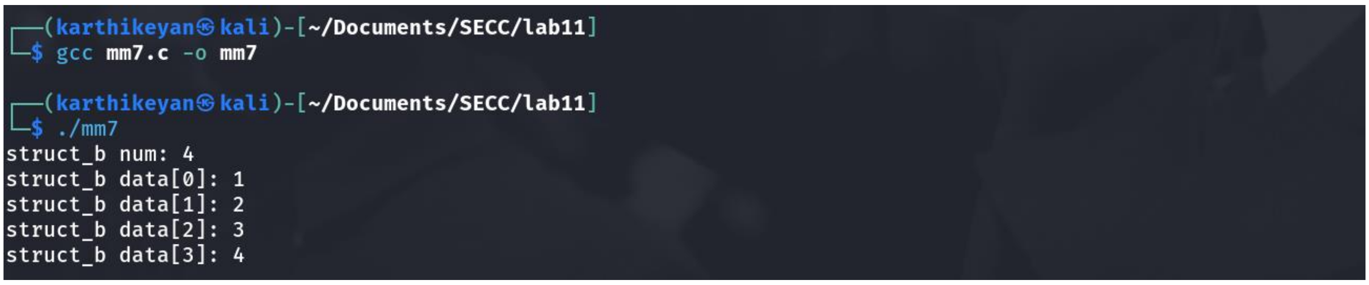
When the structure is copied, the size of the flexible array member is not considered, and only the first member of the structure, num, is copied, leaving the array contents untouched.

Compliant Solution (Copying)

This compliant solution uses memcpy() to properly copy the content of struct_a into struct_b:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void func(struct flex_array_struct *struct_a, struct flex_array_struct *struct_b) {
 if (struct_a->num > struct_b->num) {
```

```
 return;
 }
 memcpy(struct_b, struct_a, sizeof(struct flex_array_struct) + (sizeof(int) * struct_a->num));
}
int main() {
 size_t array_size_a = 4, array_size_b = 4;
 struct flex_array_struct *struct_a = malloc(sizeof(struct flex_array_struct) + sizeof(int) *
array_size_a);
 struct flex_array_struct *struct_b = malloc(sizeof(struct flex_array_struct) + sizeof(int) *
array_size_b);
 if (struct_a == NULL || struct_b == NULL) {
 return EXIT_FAILURE;
 }
 struct_a->num = array_size_a;
 for (size_t i = 0; i < array_size_a; ++i) {
 struct_a->data[i] = i + 1;
 }
 struct_b->num = array_size_b;
 func(struct_a, struct_b);
 printf("struct_b num: %zu\n", struct_b->num);
 for (size_t i = 0; i < struct_b->num; ++i) {
 printf("struct_b data[%zu]: %d\n", i, struct_b->data[i]);
 }
 free(struct_a);
 free(struct_b);
 return EXIT_SUCCESS;
}
```

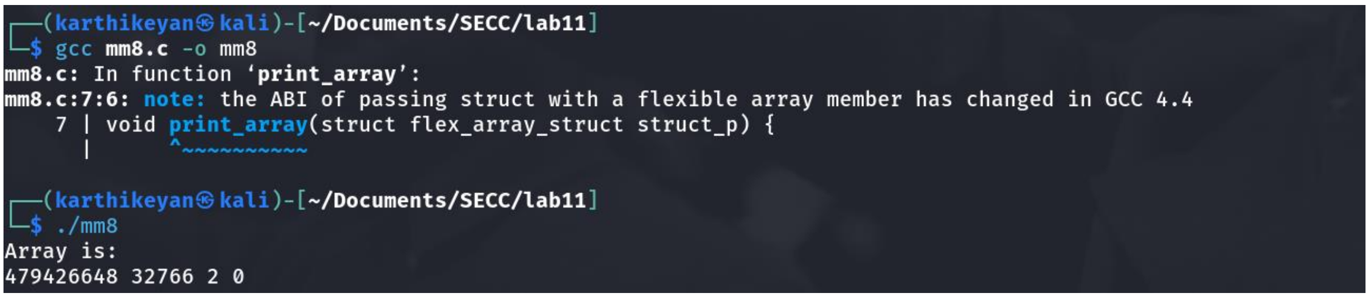


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm7.c -o mm7

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm7
struct_b num: 4
struct_b data[0]: 1
struct_b data[1]: 2
struct_b data[2]: 3
struct_b data[3]: 4
```

Noncompliant Code Example (Function Arguments)

In this noncompliant code example, the flexible array structure is passed by value to a function that prints the array elements:

```
#include <stdio.h>
#include <stdlib.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void print_array(struct flex_array_struct struct_p) {
 puts("Array is: ");
 for (size_t i = 0; i < struct_p.num; ++i) {
 printf("%d ", struct_p.data[i]);
 }
 putchar('\n');
}
void func(void) {
 struct flex_array_struct *struct_p;
 size_t array_size = 4;
 struct_p = (struct flex_array_struct *)malloc(sizeof(struct flex_array_struct) + sizeof(int) *
array_size);
 if (struct_p == NULL) {
 return;
 }
 struct_p->num = array_size;
 for (size_t i = 0; i < array_size; ++i) {
 struct_p->data[i] = i;
 }
print_array(*struct_p);
 free(struct_p);
}
int main() {
 func();
 return EXIT_SUCCESS;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm8.c -o mm8
mm8.c: In function ‘print_array’:
mm8.c:7:6: note: the ABI of passing struct with a flexible array member has changed in GCC 4.4
    7 | void print_array(struct flex_array_struct struct_p) {
      |      ^~~~~~~~~~~

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm8
Array is:
479426648 32766 2 0
```

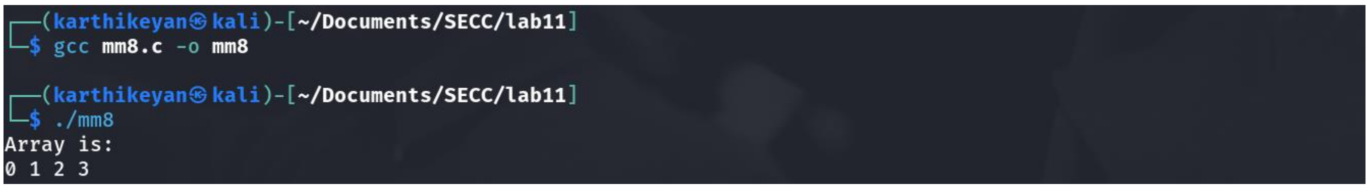
Because the argument is passed by value, the size of the flexible array member is not considered when the structure is copied, and only the first member of the structure, num, is copied.

Compliant Solution (Function Arguments)

In this compliant solution, the structure is passed by reference and not by value:

```
#include <stdio.h>
#include <stdlib.h>
struct flex_array_struct {
 size_t num;
 int data[];
};
void print_array(struct flex_array_struct *struct_p) {
 puts("Array is: ");
 for (size_t i = 0; i < struct_p->num; ++i) {
 printf("%d ", struct_p->data[i]);
 }
 putchar('\n');
}
void func(void) {
 struct flex_array_struct *struct_p;
 size_t array_size = 4;
 struct_p = (struct flex_array_struct *)malloc(sizeof(struct flex_array_struct) + sizeof(int) *
```

```
array_size);
 if (struct_p == NULL) {
 return;
 }
 struct_p->num = array_size;
 for (size_t i = 0; i < array_size; ++i) {
 struct_p->data[i] = i;
 }
 print_array(struct_p);
 free(struct_p);
}
int main() {
 func();
 return EXIT_SUCCESS;
}
```



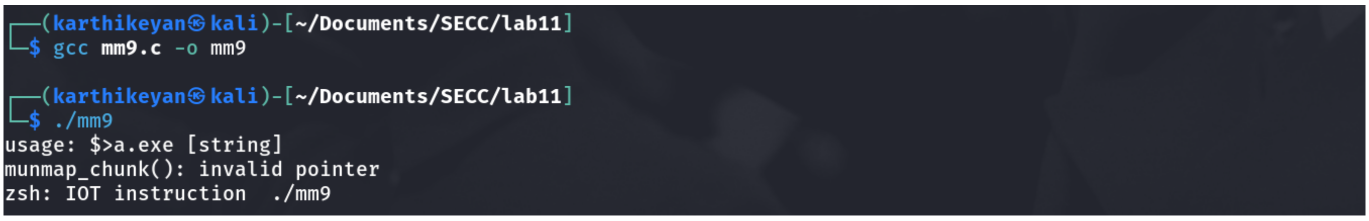
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm8.c -o mm8

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm8
Array is:
0 1 2 3
```

MEM34-C. Only free memory allocated dynamically

Noncompliant Code Example

This noncompliant code example sets c_str to reference either dynamically allocated memory or a statically allocated string literal depending on the value of argc. In either case, c_str is passed as an argument to free(). If anything other than dynamically allocated memory is referenced by c_str, the call to free(c_str) is erroneous.

```
#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) {
 printf("Error: Input string exceeds maximum allocation size\n");
return EXIT_FAILURE;
 }
 c_str = (char *)malloc(len);
 if (c_str == NULL) {
 printf("Error: Memory allocation failed\n");
 return EXIT_FAILURE;
 }
 strcpy(c_str, argv[1]);
 printf("Copied string: %s\n", c_str);
 } else {
 c_str = "usage: $>a.exe [string]";
 printf("%s\n", c_str);
 }
 free(c_str);
 return 0;
}
```

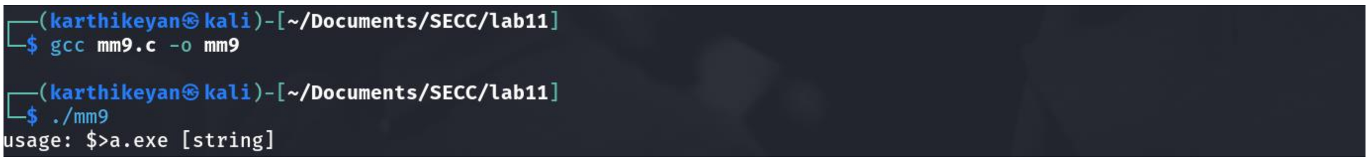


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm9.c -o mm9

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm9
usage: $>a.exe [string]
munmap_chunk(): invalid pointer
zsh: IOT instruction  ./mm9
```

Compliant Solution

This compliant solution eliminates the possibility of c_str referencing memory that is not allocated dynamically when passed to 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) {

 printf("Error: Input string exceeds maximum allocation size\n");

 return EXIT_FAILURE;

 }

 c_str = (char *)malloc(len);

 if (c_str == NULL) {

 printf("Error: Memory allocation failed\n");

 return EXIT_FAILURE;

 }

 strcpy(c_str, argv[1]);

 printf("Copied string: %s\n", c_str);

 } else {

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

 return EXIT_FAILURE;

 }

 free(c_str);

 return EXIT_SUCCESS;

}
```

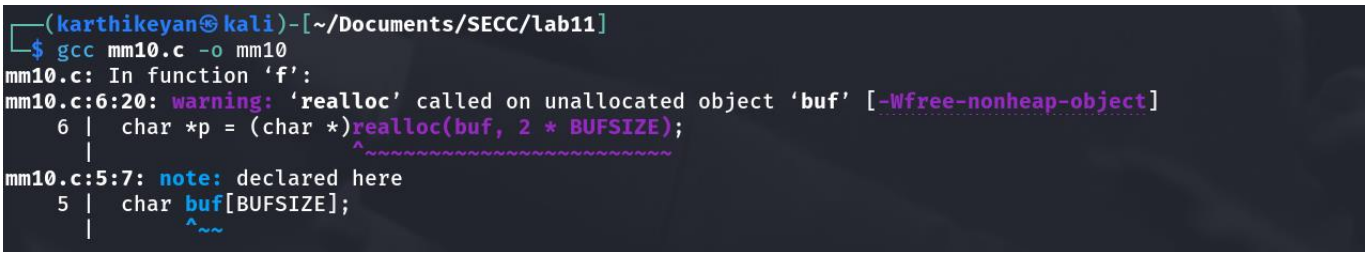


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm9.c -o mm9

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm9
usage: $>a.exe [string]
```

Noncompliant Code Example (realloc())

In this noncompliant example, the pointer parameter to realloc(), buf, does not refer to dynamically allocated memory:

```
#include <stdlib.h>

#include <stdio.h>

enum { BUFSIZE = 256 };

void f(void) {

 char buf[BUFSIZE];

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

 if (p == NULL) {

 printf("Error: Memory allocation failed\n");

 return;

 }

 // Use p as needed...

 free(p);

}

int main() {

 f();

 return EXIT_SUCCESS;

}
```



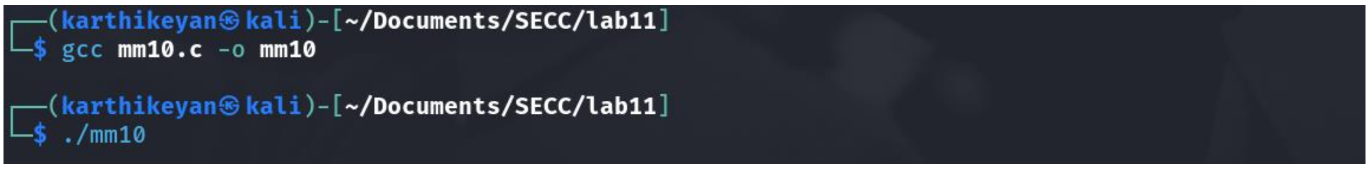
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm10.c -o mm10
mm10.c: In function ‘f’:
mm10.c:6:20: warning: ‘realloc’ called on unallocated object ‘buf’ [-Wfree-nonheap-object]
    6 |   char *p = (char *)realloc(buf, 2 * BUFSIZE);
      |                    ^~~~~~~~~~~~~~~~~~~~~~~~~~
mm10.c:5:7: note: declared here
    5 |   char buf[BUFSIZE];
      |        ^~~
```

Compliant Solution (realloc())

In this compliant solution, buf refers to dynamically allocated memory:

```
#include <stdlib.h>

#include <stdio.h>
```

```c
enum { BUFSIZE = 256 };
void f(void) {
 char *buf = (char *)malloc(BUFSIZE * sizeof(char));
 if (buf == NULL) {
 printf("Error: Memory allocation failed\n");
 return;
 }
 char *p = (char *)realloc(buf, 2 * BUFSIZE);
 if (p == NULL) {
 free(buf);
 printf("Error: Memory reallocation failed\n");
 return;
 }
 buf = p;
 // Use buf as needed...
 free(buf);
}
int main() {
 f();
 return EXIT_SUCCESS;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm10.c -o mm10

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm10
```

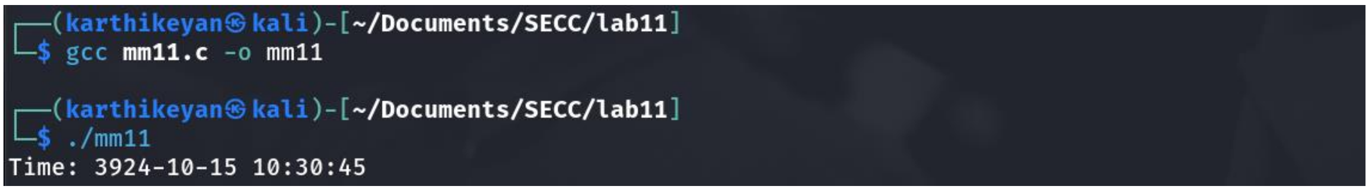
Note that realloc() will behave properly even if malloc() failed, because when given a null pointer, realloc() behaves like a call to malloc().

MEM35-C. Allocate sufficient memory for an object

Noncompliant Code Example (Pointer)

In this noncompliant code example, inadequate space is allocated for a struct tm object because the size of the pointer is being used to determine the size of the pointed-to object:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
struct tm *make_tm(int year, int mon, int day, int hour, int min, int sec) {
 struct tm *tmb;
 tmb = (struct tm *)malloc(sizeof(struct tm));
 if (tmb == NULL) {
 return NULL;
 }
 *tmb = (struct tm) {
 .tm_sec = sec, .tm_min = min, .tm_hour = hour,
 .tm_mday = day, .tm_mon = mon, .tm_year = year
 };
 return tmb;
}
int main() {
 struct tm *time_info = make_tm(2024, 9, 15, 10, 30, 45);
 if (time_info != NULL) {
 printf("Time: %d-%02d-%02d %02d:%02d:%02d\n",
 time_info->tm_year + 1900, time_info->tm_mon + 1,
 time_info->tm_mday, time_info->tm_hour,
 time_info->tm_min, time_info->tm_sec);
 free(time_info);
 } else {
 printf("Error: Memory allocation failed\n");
 }
 return EXIT_SUCCESS;
}
```

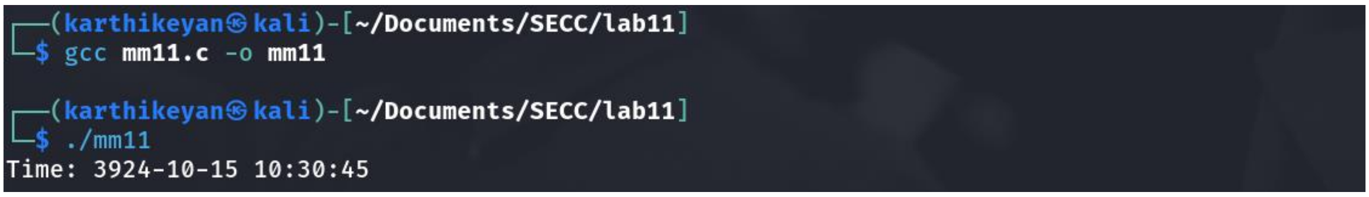


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm11.c -o mm11

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm11
Time: 3924-10-15 10:30:45
```

Compliant Solution (Pointer)

In this compliant solution, the correct amount of memory is allocated for the struct tm object. When allocating space for a single object, passing the (dereferenced) pointer type to the sizeof operator is a simple way to allocate sufficient memory. Because the sizeof operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
struct tm *make_tm(int year, int mon, int day, int hour, int min, int sec) {
 struct tm *tmb;
 tmb = (struct tm *)malloc(sizeof(*tmb));
if (tmb == NULL) {
 return NULL;
 }
 *tmb = (struct tm) {
 .tm_sec = sec, .tm_min = min, .tm_hour = hour,
 .tm_mday = day, .tm_mon = mon, .tm_year = year
 };
 return tmb;
}
int main() {
 struct tm *time_info = make_tm(2024, 9, 15, 10, 30, 45);
 if (time_info != NULL) {
 printf("Time: %d-%02d-%02d %02d:%02d:%02d\n",
 time_info->tm_year + 1900, time_info->tm_mon + 1,
 time_info->tm_mday, time_info->tm_hour,
 time_info->tm_min, time_info->tm_sec);
 free(time_info);
 } else {
 printf("Error: Memory allocation failed\n");
 }
 return EXIT_SUCCESS;
}
```



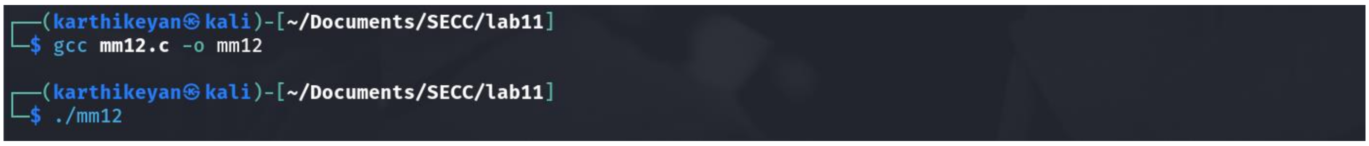
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm11.c -o mm11

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm11
Time: 3924-10-15 10:30:45
```

Noncompliant Code Example (Integer)

In this noncompliant code example, an array of long is allocated and assigned to p. The code attempts to check for unsigned integer overflow in compliance with INT30-C. Ensure that unsigned integer operations do not wrap and also ensures that len is not equal to zero. (See MEM04-C. Beware of zero-length allocations.) However, because sizeof(int) is used to compute the size, and not sizeof(long), an insufficient amount of memory can be allocated on implementations where sizeof(long) is larger than sizeof(int), and filling the array can cause a heap buffer overflow.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
void function(size_t len) {
long *p;
if (len == 0 || len > SIZE_MAX / sizeof(long)) {
printf("Error: Length is either zero or exceeds maximum allowable size.\n");
return;
}
p = (long *)malloc(len * sizeof(long));
if (p == NULL) {
printf("Error: Memory allocation failed\n");
```

```
 return;
 }
 // Use p as needed...
 free(p);
}
int main() {
 function(10); // Example usage
 return EXIT_SUCCESS;
}
```



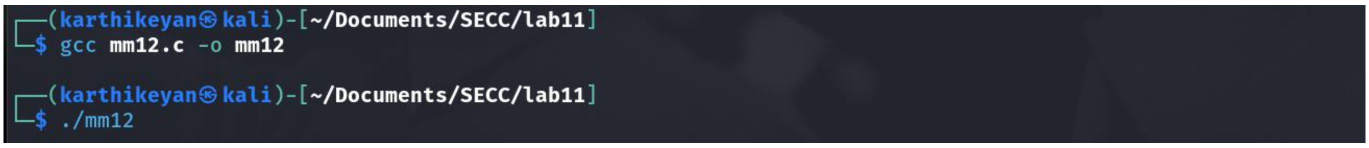
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm12.c -o mm12

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm12
```

Compliant Solution (Integer)

This compliant solution uses sizeof(long) to correctly size the memory allocation:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
void function(size_t len) {
 long *p;
 if (len == 0 || len > SIZE_MAX / sizeof(long)) {
 printf("Error: Length is either zero or exceeds maximum allowable size.\n");
 return;
 }
 p = (long *)malloc(len * sizeof(long));
 if (p == NULL) {
 printf("Error: Memory allocation failed\n");
 return;
 }
 // Use p as needed...
 free(p);
}
```

```
int main() {
 function(10); // Example usage
 return EXIT_SUCCESS;
}
```

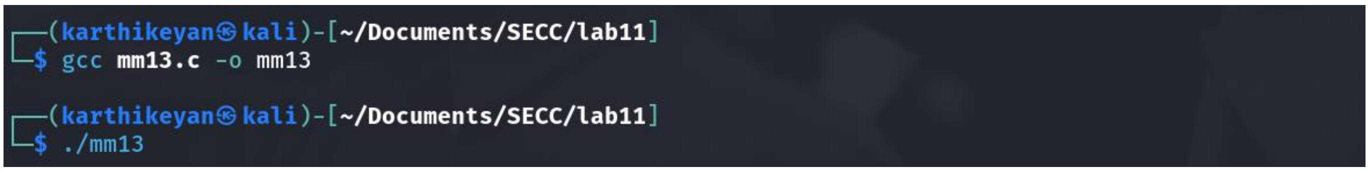


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm12.c -o mm12

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm12
```

Compliant Solution (Integer)

Alternatively, sizeof(*p) can be used to properly size the allocation:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
void function(size_t len) {
 long *p;
 if (len == 0 || len > SIZE_MAX / sizeof(*p)) {
 printf("Error: Length is either zero or exceeds maximum allowable size.\n");
 return;
 }
 p = (long *)malloc(len * sizeof(*p));
 if (p == NULL) {
 printf("Error: Memory allocation failed\n");
 return;
 }
 // Use p as needed...
 free(p);
}
int main() {
 function(10); // Example usage
 return EXIT_SUCCESS;
}
```



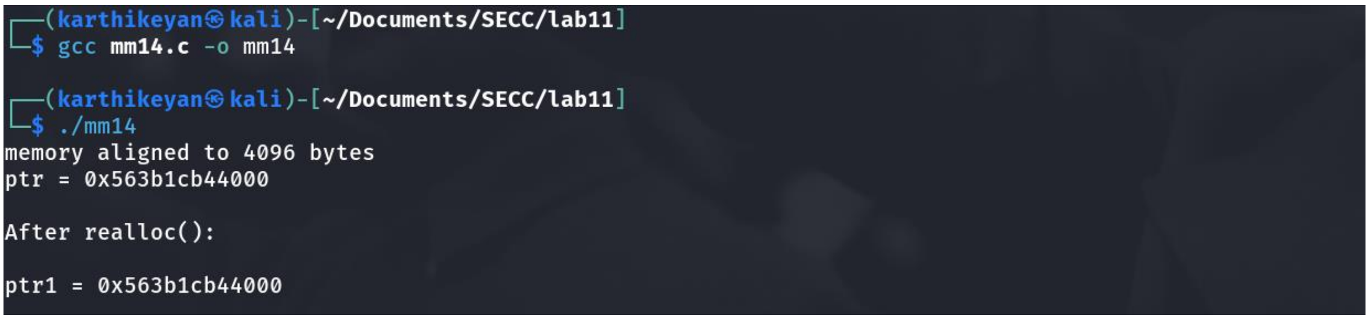
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm13.c -o mm13

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm13
```

MEM36-C. Do not modify the alignment of objects by calling realloc()

Noncompliant Code Example

This noncompliant code example returns a pointer to allocated memory that has been aligned to a 4096-byte boundary. If the resize argument to the realloc() function is larger than the object referenced by ptr, then realloc() will allocate new memory that is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement but may not preserve the stricter alignment of the original object.

```
#include <stdlib.h>

#include <stdio.h>

int main(void) {

 size_t size = 16;

 size_t resize = 1024;

 size_t align = 1 << 12;

 int *ptr;

 int *ptr1;

 if (posix_memalign((void **)&ptr, align , size) != 0) {

 exit(EXIT_FAILURE);

 }

 printf("memory aligned to %zu bytes\n", align);

 printf("ptr = %p\n\n", ptr);

 if ((ptr1 = (int*) realloc((int *)ptr, resize)) == NULL) {

 exit(EXIT_FAILURE);

 }

 puts("After realloc(): \n");

 printf("ptr1 = %p\n", ptr1);

 free(ptr1);

 return 0;

}
```

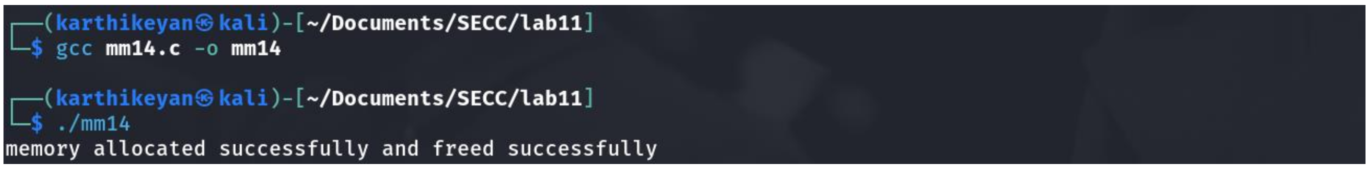


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm14.c -o mm14

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm14
memory aligned to 4096 bytes
ptr = 0x563b1cb44000

After realloc():

ptr1 = 0x563b1cb44000
```

Compliant Solution

This compliant solution allocates resize bytes of new memory with the same alignment as the old memory, copies the original memory content, and then frees the old memory. This solution has implementation-defined behavior because it depends on whether extended alignments in excess of _Alignof (max_align_t) are supported and the contexts in which they are supported. If not supported, the behavior of this compliant solution is undefined.

```
#include <stdlib.h>

#include <string.h>

#include <stdio.h>

void func(void) {

 size_t resize = 1024;

 size_t alignment = 1 << 12;

 int *ptr;

 int *ptr1;


 if (NULL == (ptr = (int *)aligned_alloc(alignment, sizeof(int)))) {

 printf("Error: Memory allocation for ptr failed\n");

 return;

 }


 if (NULL == (ptr1 = (int *)aligned_alloc(alignment, resize))) {

 printf("Error: Memory allocation for ptr1 failed\n");

 free(ptr);

 return;

 }
 if (NULL == memcpy(ptr1, ptr, sizeof(int))) {
 printf("Error: Memory copy failed\n");
 free(ptr);
 free(ptr1);
 return;
 }
 free(ptr);
 free(ptr1);
}
int main() {
 func(); // Example usage
 printf("memory allocated successfully and freed successfully\n");
 return EXIT_SUCCESS;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ gcc mm14.c -o mm14

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab11]
└─$ ./mm14
memory allocated successfully and freed successfully
```