

# UNIT-3—Dynamic Memory Management

## Memory Managers

In C++, memory managers are responsible for allocating and deallocating memory dynamically during program execution.

These memory management operations are crucial for the efficient use of system resources, and they play a key role in ensuring that memory is used effectively, particularly when dealing with large or complex applications.

C++ provides various mechanisms and tools for managing memory, including custom memory managers, standard memory management facilities (like `new`, `delete`, `malloc`, `free`), and smart pointers.

A memory manager can be seen as an abstraction that helps allocate memory (usually on the heap) and ensure proper deallocation, preventing issues such as memory leaks, dangling pointers, and double frees.

# UNIT-3—Dynamic Memory Management

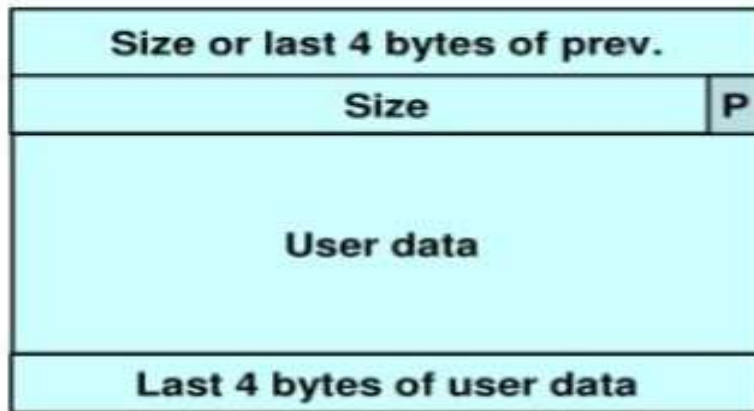
## Memory Managers

- **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.

# UNIT-3—Dynamic Memory Management

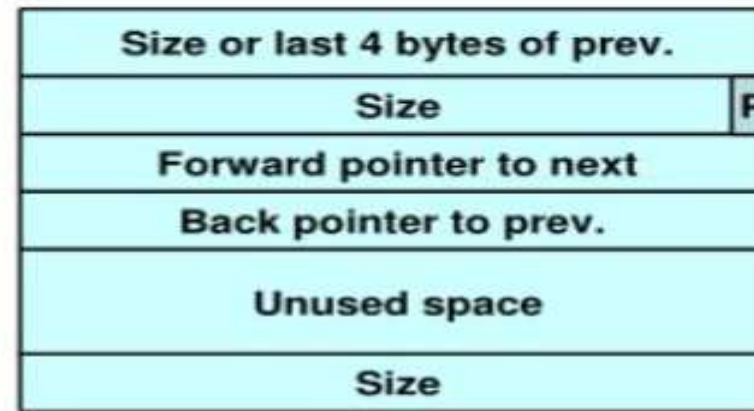
## Doug Lea's Memory Allocator(dlmalloc) :

It is a native version of malloc. Doug Lea's malloc manages the heap and provides standard memory management. In dlmalloc memory chunks are either allocated to process or are free. In the figure P indicates PREV\_INUSE bit to indicate whether or not free chunk allocated.



**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.

**Structures of allocated and free chunks**

## UNIT-3—Dynamic Memory Management

### Doug Lea's Memory Allocator(dlmalloc) :

- 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,
  - enables adjacent free chunks to be consolidated to avoid fragmentation of memory.

Structure of allocated and free chunks

# UNIT-3—Dynamic Memory Management

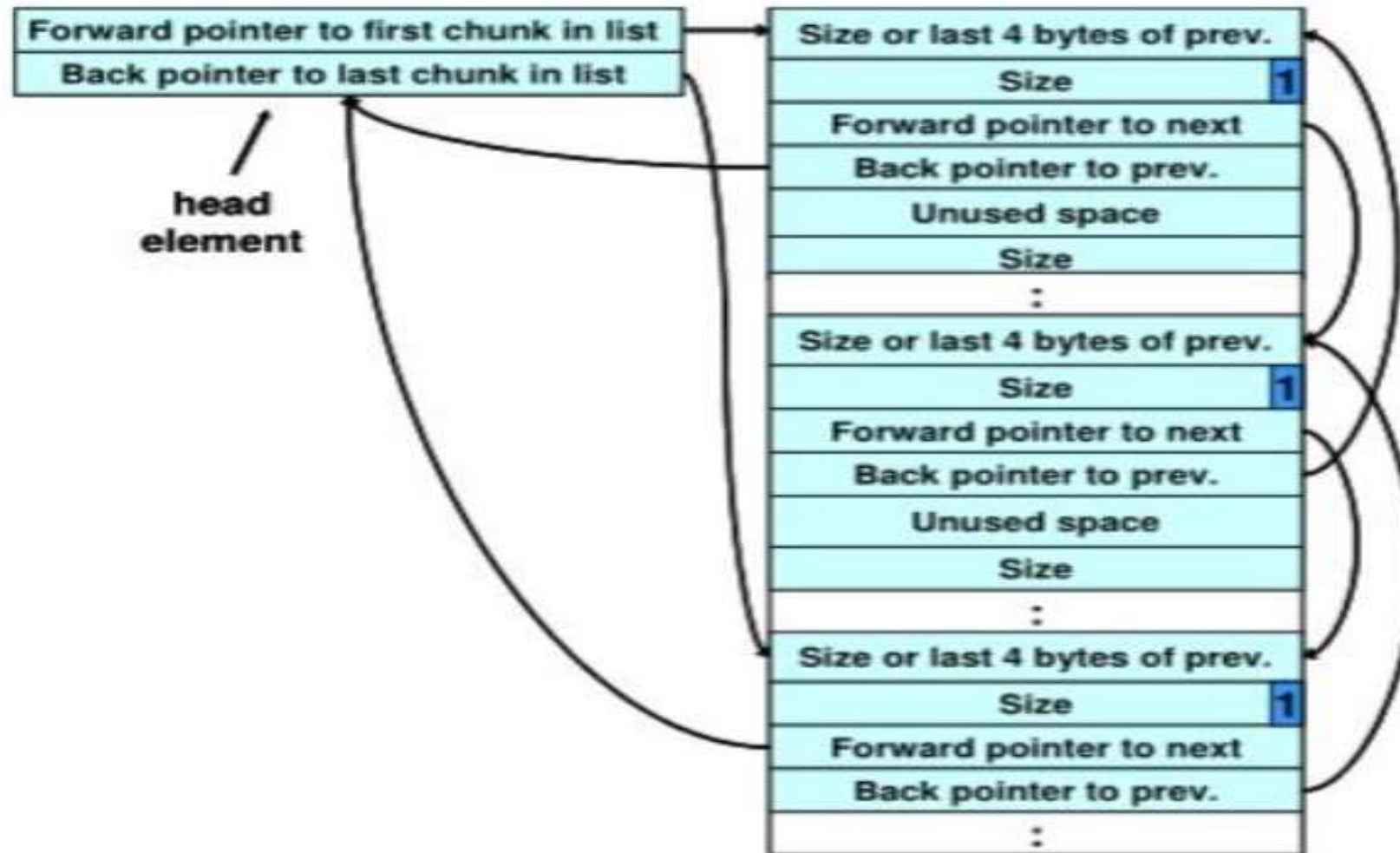


Fig: Free list double –linked structure

## UNIT-3—Dynamic Memory Management

### Doug Lea's Memory Allocator(dlmalloc) :

- 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.



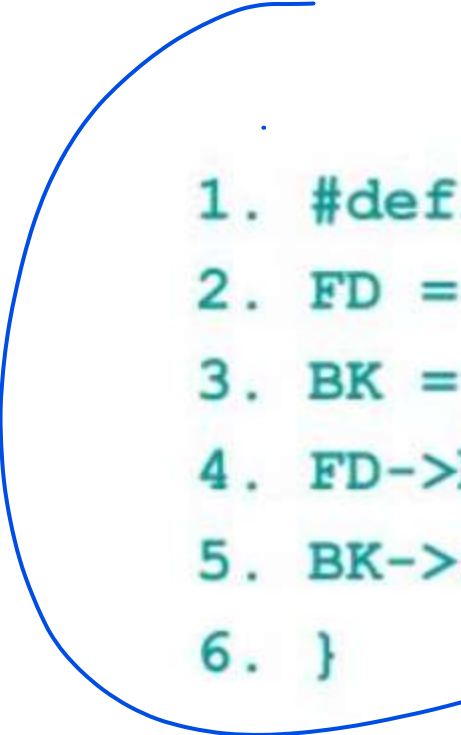
## UNIT-3—Dynamic Memory Management

### Doug Lea's Memory Allocator(dlmalloc) :

- 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.

## UNIT-3—Dynamic Memory Management

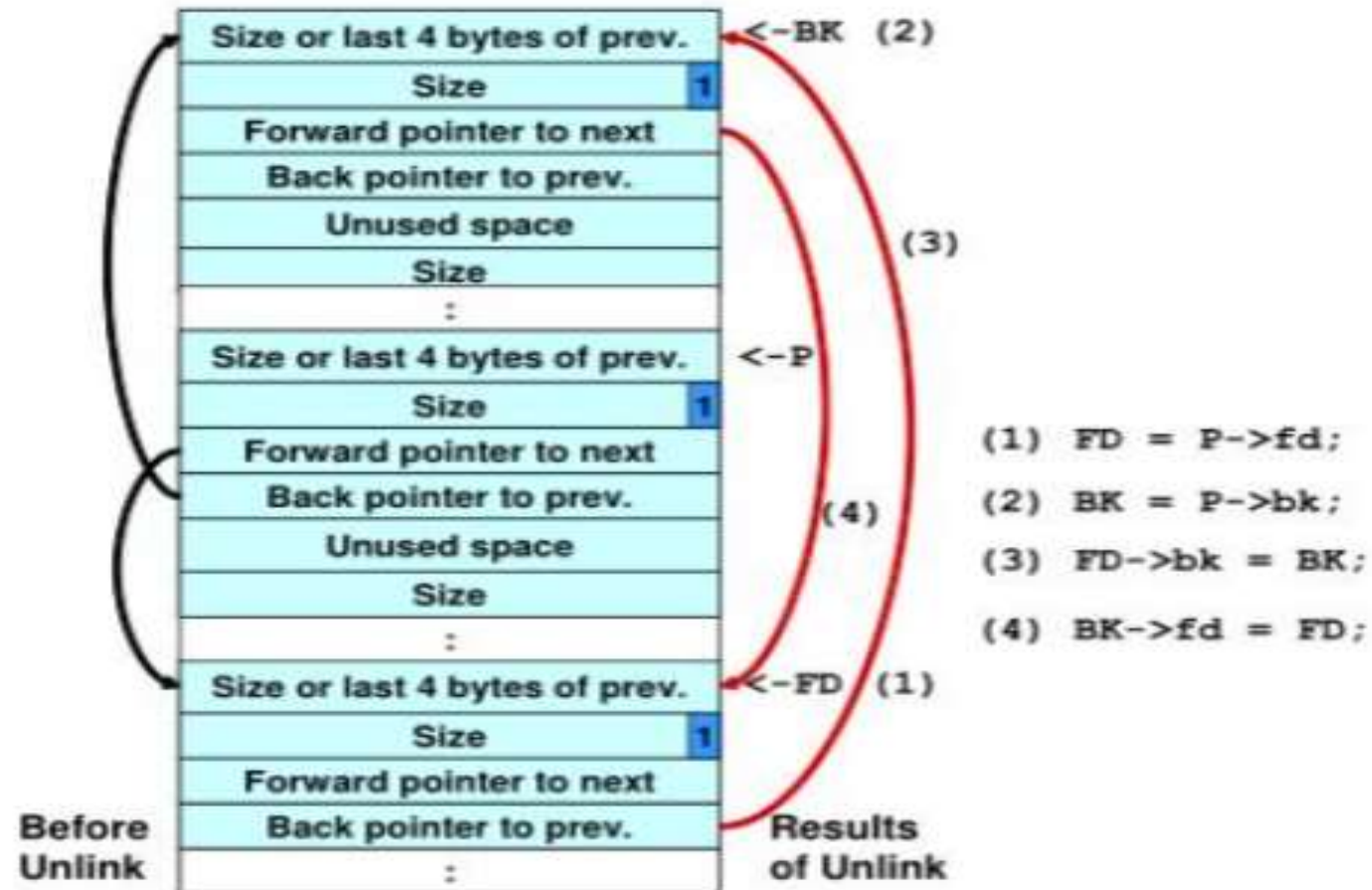
### Doug Lea's Memory Allocator(dlmalloc) :



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



# UNIT-3—Dynamic Memory Management



### Buffer Overflows on Heap

- 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.

## UNIT-3—Dynamic Memory Management

### Unlink Technique:

The unlink technique is used to exploit a buffer overflow to manipulate the boundary tags on chunks of memory to trick the `unlink()` macro into writing 4 bytes of data to an arbitrary location.

# UNIT-3—Dynamic Memory Management

## Unlink Technique:

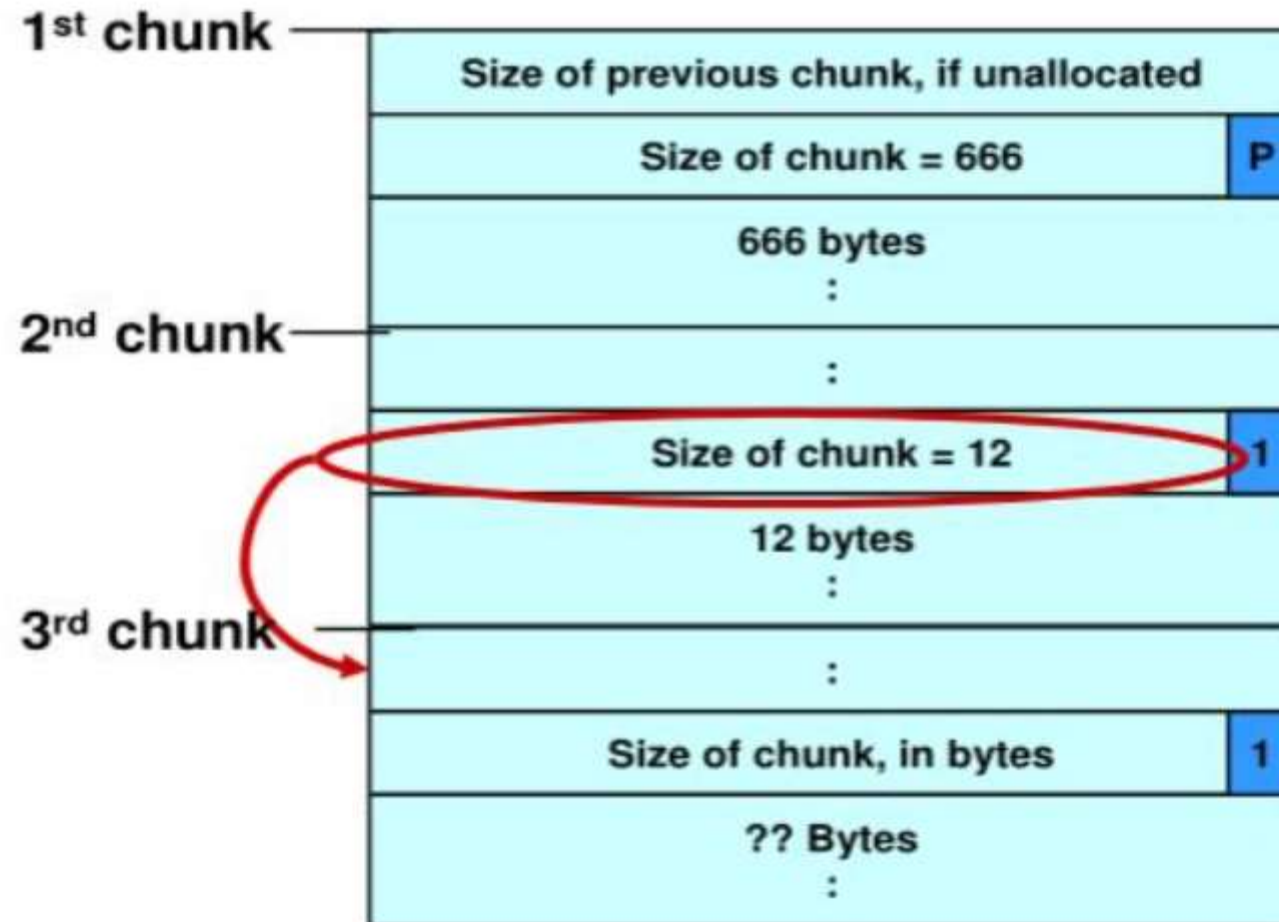
```
1. #include <stdlib.h>
2. #include <string.h>
3. int main(int argc, char *argv[]) {
4.     char *first, *second, *third;
5.     first = malloc(666);
6.     second = malloc(12);
7.     third = malloc(12);
8.     strcpy(first, argv[1]);
9.     free(first);
10.    free(second);
11.    free(third);
12.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

Memory allocation  
chunk 1

Memory allocation  
chunk 2

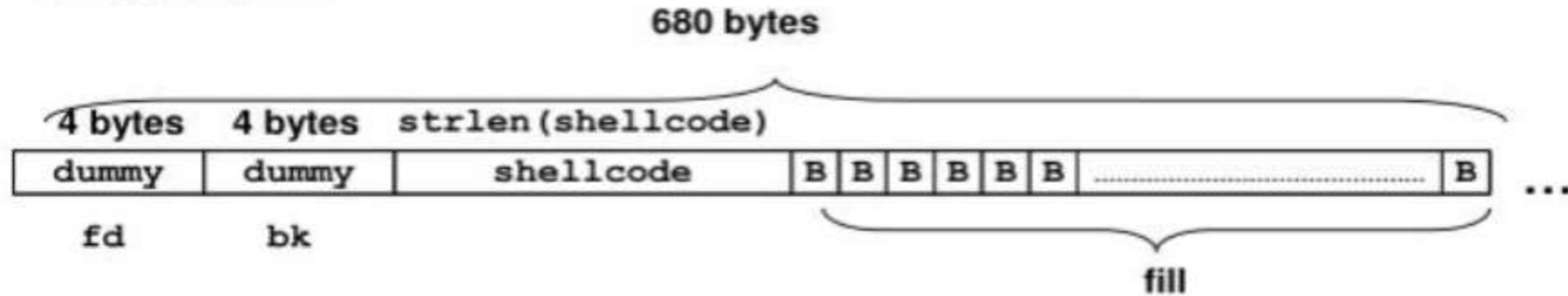
Memory allocation  
chunk 3

# UNIT-3—Dynamic Memory Management

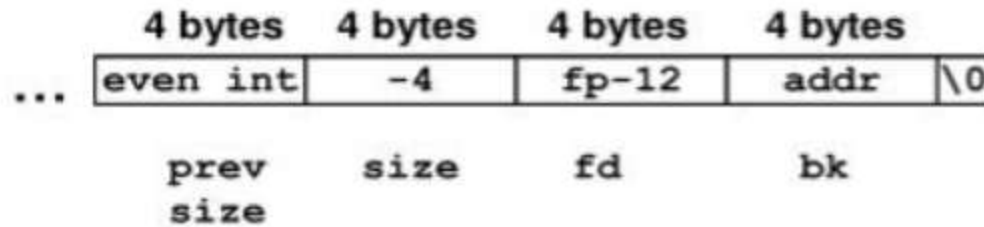


# UNIT-3—Dynamic Memory Management

## First Chunk



## Second Chunk





## UNIT-2—Dynamic Memory Management

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

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

## UNIT-2—Dynamic Memory Management

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`

## **Null Pointers:**

One obvious technique to reduce vulnerabilities in C and C++ programs is to set pointers to NULL after the referenced memory is deallocated.

Dangling pointers can result in writing to freed memory and double free vulnerabilities. Any attempt to dereference the pointer will result in a fault, which increases the likelihood that the error is detected during the implementation and test.

If the pointer is set to null memory can be freed multiple times without consequence.

## **Consistent Memory Management Conventions:**

Following Conventions could be applied:

- Use the same pattern for allocating and freeing memory
- Allocate and free memory in the same module at the same level of abstraction
- Match allocations and deallocations

## phkmalloc

PHKmalloc is a memory allocator designed for efficient and predictable memory management, especially for high-performance applications. It focuses on reducing memory fragmentation, minimizing the time spent on allocation/deallocation, and optimizing cache usage.

Key Features of PHKmalloc:

- **Efficient Block Management:** PHKmalloc reduces fragmentation by allocating memory in blocks and chunks of predefined sizes, which can be reused efficiently.
- **Garbage Collection (Optional):** In some configurations, PHKmalloc may include garbage collection mechanisms to prevent long-term fragmentation.
- **Performance Optimizations:** It minimizes overhead by reducing system calls and memory management-related locks.

## Randomization:

Randomization works on the principle that it is harder to hit a moving target than a still target.

Randomizing the addresses of blocks of memory returned by the memory manager can make it more difficult to exploit heap based vulnerability.

Randomizing the memory addresses can occur in multiple locations. For both the windows and Unix Operating System, the memory manager requests the memory pages from the operating system, randomize both pages returned by the operating system.



## OpenBSD:

OpenBSD Unix Variant was designed with a additional emphasis on security.

Open BSD adopted phkmalloc and adopted it to support randomization and guard pages(unmapped pages placed between all allocations of the memory the size of one page or larger to detect overflow.

# Mitigation Strategies

jemalloc (short for "JavaScript Memory Allocator" originally, but now a general-purpose memory allocator) is a high-performance memory allocator used in various systems and applications, especially where memory fragmentation and efficiency are critical.

It was designed to address inefficiencies in traditional allocators (such as malloc and free in C/C++), offering better performance, especially for multi-threaded applications.

Several versions of jemalloc are available for FreeBSD, Linux, Windows and macOS, Mozilla Firefox