

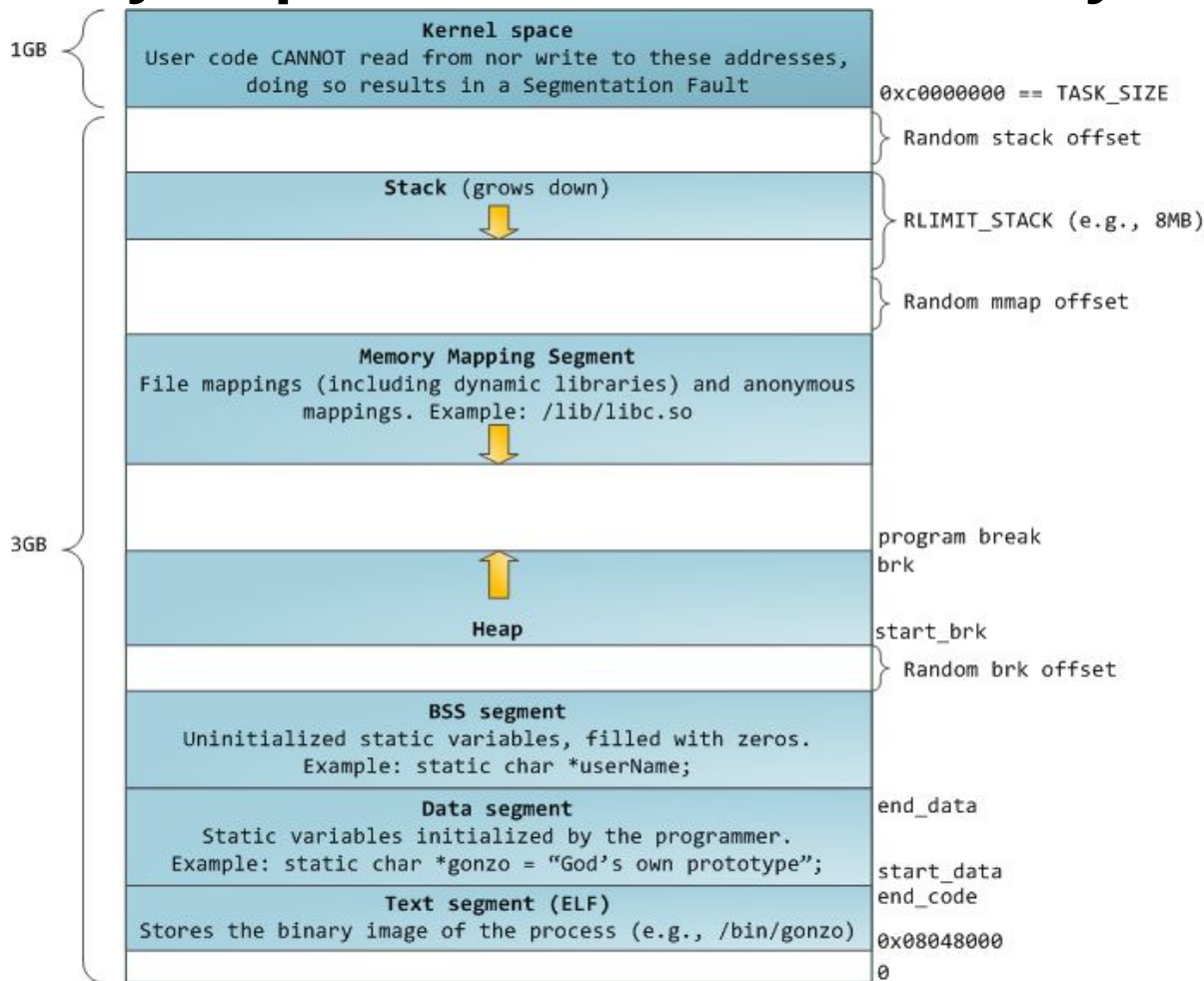
# **CSE 410/565: Computer Security**

Instructor: Dr. Ziming Zhao

# Today

1. Heap and heap exploitation

# Memory Map of Linux Process (32 bit system)



<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

# The Heap

The heap is pool of memory used for dynamic allocations at runtime. Heap memory is different from stack memory in that it is ***persistent between functions***.

- **malloc()** grabs memory on the heap; keyword ***new*** in C++
- **free()** releases memory on the heap; keyword ***delete*** in C++

Both are standard C library interfaces. Neither of them directly maps to a system call.

# malloc() and free()

```
void* malloc(size_t size);
```

Allocates `size` bytes of uninitialized storage. If allocation succeeds, returns a pointer that is suitably aligned for any object type with fundamental alignment.

```
void free(void* ptr);
```

Deallocates the space previously allocated by `malloc()`, etc.

# calloc() and realloc()

```
void *calloc(size_t nitems, size_t size)
```

The difference in malloc and calloc is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero.

```
void *realloc(void *ptr, size_t size)
```

Resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

# How to use malloc() and free()

```
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf("Hello %s!\n", buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```

# Heap vs. Stack

## Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things

## Slower, Manual

- Done by the programmer
- malloc/calloc/realloc/free
- new/delete

## Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args

## Fast, Automatic; Done by the compiler

- Abstracts away any concept of allocating/de-allocating



# Heap Implementations

**Doug Lea malloc** or **dlmalloc**. Default native version of malloc in some old distributions of Linux (<http://gee.cs.oswego.edu/dl/html/malloc.html>)

**ptmalloc**. ptmalloc is based on dmalloc and was extended for use with multiple threads. On Linux systems, ptmalloc has been put to work for years as part of the GNU C library.

**tcmalloc**. Google's customized implementation of C's malloc() and C++'s operator new (<https://github.com/google/tcmalloc>)

**jemalloc**. jemalloc is a general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support.

The **Hoard** memory allocator. UMass Amherst CS Professor Emery Berger

# Which implementation on my laptop?

ldd --version

GLIBC 2.31

Ptmalloc2

<https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c>

```
→ heapfreees ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.2) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

# Overview of dlmalloc

The Linux version of the dynamic memory allocator. Even though it has been updated, from the point of view of software infused bugs and exploits, new versions are still more or less similar to the original one.

Design goals:

**Maximizing Portability** To rely on as few system-dependent features as possible, system calls in particular.

**Minimizing Space** The allocator should not waste memory. It should obtain the least amount of memory from the system it requires, and should maintain memory in ways that minimize fragmentation—that is, it should try to avoid creating a large number of contiguous chunks of memory that are not used by the program.

**Minimizing Time** The malloc(), free(), and realloc calls should be fast on average.

# Overview of dmalloc

The Linux version of the dynamic memory allocator. Even though it has been updated, from the point of view of software infused bugs and exploits, new versions are still more or less similar to the original one.

Design goals:

**Maximizing Locality** Allocate chunks of memory that are typically requested or used together near each other. This will help minimize CPU page and cache misses.

**Maximizing Error Detection** Should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on. It is not supposed to work as a general memory leak detection tool at the cost of slowing down.

**Minimizing Anomalies** It should have reasonably similar performance characteristics across a wide range of possible applications whether they are GUI or server programs, string processing applications, or network tools.

# Malloc\_chunk (ptmalloc2 in glibc2.31)

```
struct malloc_chunk { Both in-use and freed  
    INTERNAL_SIZE_T    mchunk_prev_size; /* Size of previous chunk (if free). */  
    INTERNAL_SIZE_T    mchunk_size;     /* Size in bytes, including overhead. */  
  
    struct malloc_chunk* fd;             /* double links -- used only if free. */  
    struct malloc_chunk* bk;             /* double links -- used only if free. */  
  
    /* Only used for large blocks: pointer to next larger size. */  
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */  
    struct malloc_chunk* bk_nextsize;  
};
```

**INTERNAL\_SIZE\_T** is the same as size\_t. 8 bytes in 64 bit;  
4 bytes in 32 bits machine.  
Pointer is 8/4 bytes on a 64/32 bit machine, respectively.

# Heap Chunks (figures in 32 bit)

```
buffer = malloc(0x100);
```

```
//Out comes a heap chunk
```

**Previous Chunk Size:** Size of previous chunk (if prev chunk is free)

**Chunk Size:** Size of entire chunk including overhead

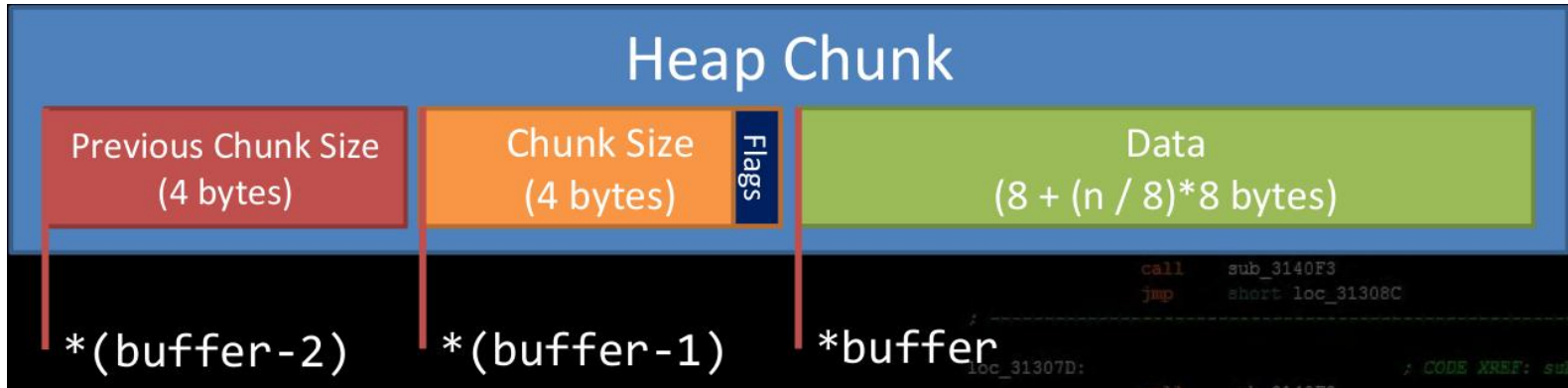
**Data:** Your newly allocated memory / ptr returned by malloc

**Flags:** Because of byte alignment, the lower 3 bits of the chunk size field would always be zero. Instead they are used for flag bits.

0x01 PREV\_INUSE – set when previous chunk is in use

0x02 IS\_MMAPPED – set if chunk was obtained with mmap()

0x04 NON\_MAIN\_ARENA – set if chunk belongs to a thread arena



# Malloc Trivia

How many bytes on the heap are your *malloc chunks* really taking up?

- `malloc(32);`
- `malloc(4);`
- `malloc(20);`
- `malloc(0);`

# code/heap sizes

```
int main()
{
    unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32};
    unsigned int * ptr[10];
    int i;

    for(i = 0; i < 10; i++)
        ptr[i] = malloc(lengths[i]);

    for(i = 0; i < 9; i++)
        printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
            lengths[i],
            (unsigned int)ptr[i],
            (ptr[i+1]-ptr[i])*sizeof(unsigned int));

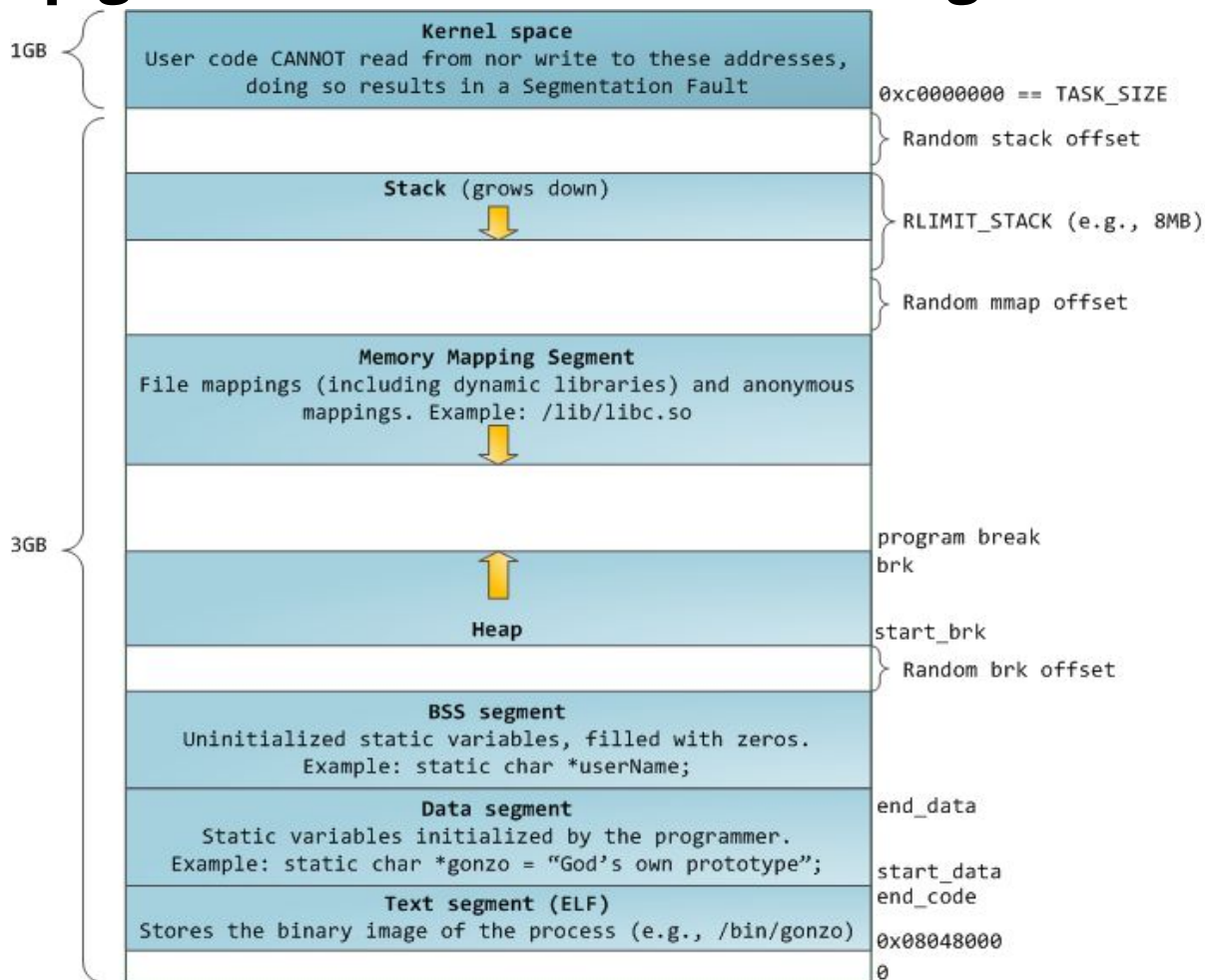
    return 0;}

```

<https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/sizes.c>



# Heap goes from low address to high address



<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

# code/heapsizes

```
int main()
{
    unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32};
    unsigned int * ptr[10];
    int i;

    for(i = 0; i < 10; i++)
        ptr[i] = malloc(lengths[i]);

    for(i = 0; i < 9; i++)
        printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
            lengths[i],
            (unsigned int)ptr[i],
            (ptr[i+1]-ptr[i])*sizeof(unsigned int));

    return 0;}

```

H



Chunk 10

...

Chunk 3

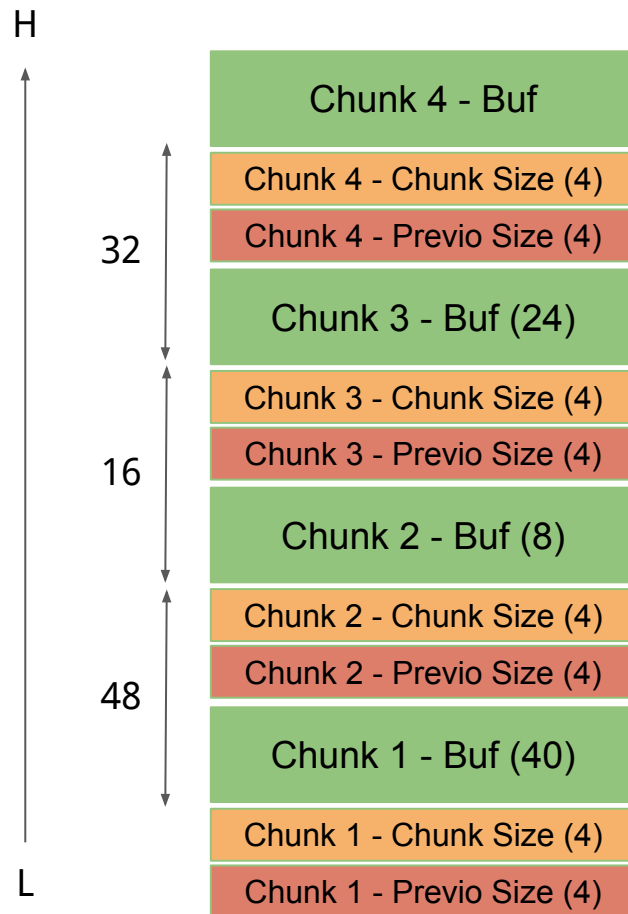
Chunk 2

L

Chunk 1

# code/heapsizes 32bit

```
→ heapsizes ./heapsizes32
malloc(32) is at 0x5695b1a0, 48 bytes to the next pointer
malloc( 4) is at 0x5695b1d0, 16 bytes to the next pointer
malloc(20) is at 0x5695b1e0, 32 bytes to the next pointer
malloc( 0) is at 0x5695b200, 16 bytes to the next pointer
malloc(64) is at 0x5695b210, 80 bytes to the next pointer
malloc(32) is at 0x5695b260, 48 bytes to the next pointer
malloc(32) is at 0x5695b290, 48 bytes to the next pointer
malloc(32) is at 0x5695b2c0, 48 bytes to the next pointer
malloc(32) is at 0x5695b2f0, 48 bytes to the next pointer
```



# code/heapsizes 64bit

```
→ heapsizes ./heapsizes
malloc(32) is at 0xc91e02a0, 48 bytes to the next pointer
malloc( 4) is at 0xc91e02d0, 32 bytes to the next pointer
malloc(20) is at 0xc91e02f0, 32 bytes to the next pointer
malloc( 0) is at 0xc91e0310, 32 bytes to the next pointer
malloc(64) is at 0xc91e0330, 80 bytes to the next pointer
malloc(32) is at 0xc91e0380, 48 bytes to the next pointer
malloc(32) is at 0xc91e03b0, 48 bytes to the next pointer
malloc(32) is at 0xc91e03e0, 48 bytes to the next pointer
malloc(32) is at 0xc91e0410, 48 bytes to the next pointer
```

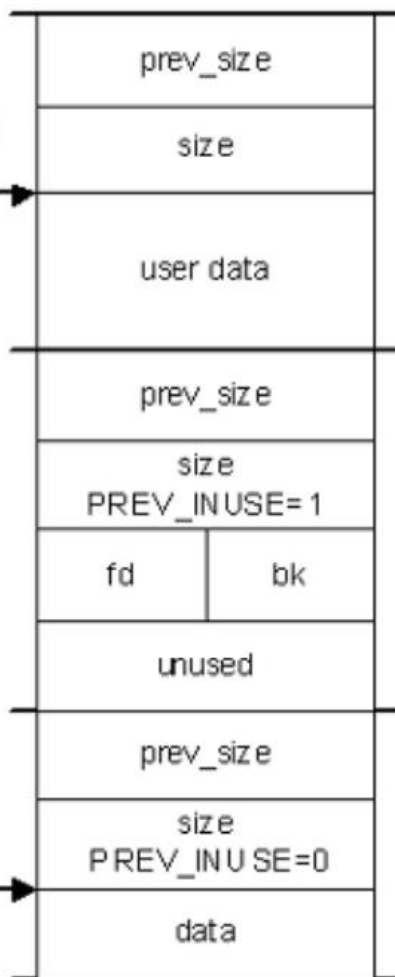
# Malloc Trivia

How many bytes on the heap are your *malloc chunks* really taking up?

- `malloc(32);` 48 bytes (32bit/64bit)
- `malloc(4);` 16 bytes (32bit) / 32 bytes (64bit)
- `malloc(20);` 32 bytes (32bit/64bit)
- `malloc(0);` 16 bytes (32bit) / 32 bytes (64bit)

chunk A,  
being freed

A →



chunk A will be  
forward consolidated  
with B

# code/heapchunks

```
void print_chunk(size_t * ptr, unsigned int len)
{
    printf("[ prev - 0x%08x ][ size - 0x%08x ][ data buffer (0x%08x) -----> ... ] - from
malloc(%d)\n", *(ptr-2), *(ptr-1), (unsigned int)ptr, len);}

int main()
{
    void * ptr[LEN];
    unsigned int lengths[] = {0, 4, 8, 16, 24, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384};
    int i;

    printf("mallocing...\n");

    for(i = 0; i < LEN; i++)
        ptr[i] = malloc(lengths[i]);

    for(i = 0; i < LEN; i++)
        print_chunk(ptr[i], lengths[i]);
    return 0;}
```

Extended from  
[https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/heap\\_chunks.c](https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/heap_chunks.c)



→ heapchunks ./heapchunks32

mallocing...

```
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665b0) -----> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665c0) -----> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665d0) -----> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x57b665e0) -----> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x57b66600) -----> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x57b66620) -----> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x57b66650) -----> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x57b666a0) -----> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x57b66730) -----> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x57b66840) -----> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x57b66a50) -----> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x57b66e60) -----> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x57b67670) -----> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x57b68680) -----> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x57b6a690) -----> ... ] - from malloc(16384)
```

→ heapchunks ./heapchunks

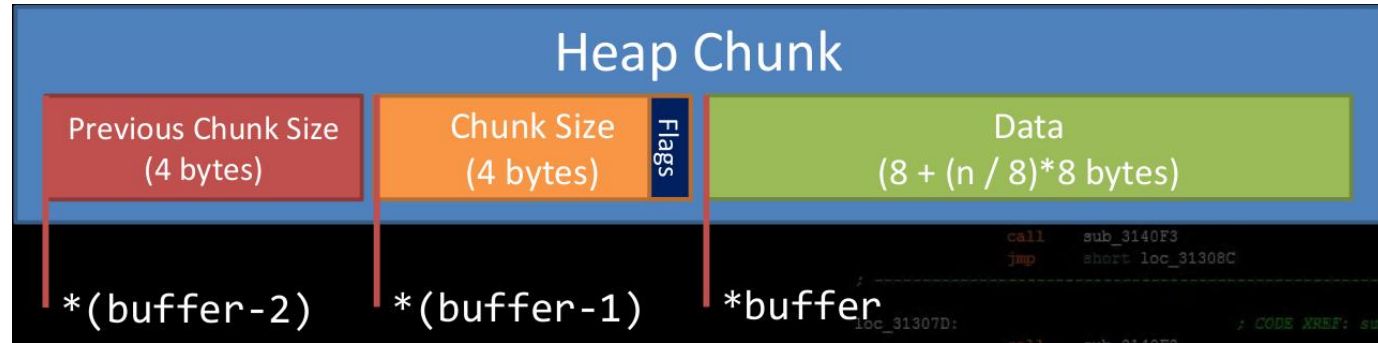
mallocing...

```
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046b0) -----> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046d0) -----> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046f0) -----> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x66504710) -----> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x66504730) -----> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x66504750) -----> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x66504780) -----> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x665047d0) -----> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x66504860) -----> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x66504970) -----> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x66504b80) -----> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x66504f90) -----> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x665057a0) -----> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x665067b0) -----> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x665087c0) -----> ... ] - from malloc(16384)
```



# Heap Chunks – Two states (figures in 32 bit)

Heap chunks exist in two states  
– in use (malloc'd)

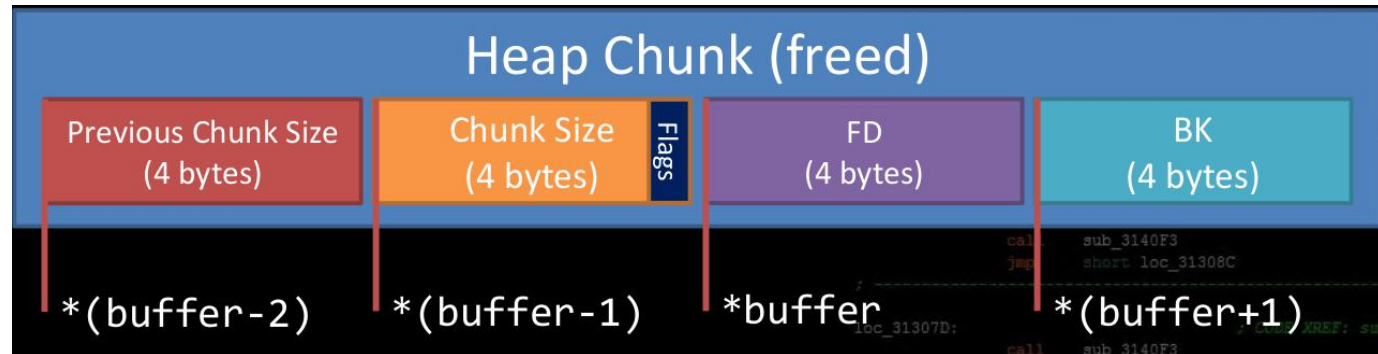


– free'd.

Forward Pointer: A pointer to the next freed chunk

Backwards Pointer: A pointer to the previous freed chunk

Implementation-defined.



# code/heapfrees

```
void print_inuse_chunk(unsigned int * ptr)
{
    printf("[ prev - 0x%08x ][ size - 0x%08x ][ data buffer  
(0x%08x) ----> ... ] - Chunk 0x%08x - In use\n", \
        *(ptr-2),
        *(ptr-1),
        (unsigned int)ptr,
        (unsigned int)(ptr-2));
}

void print_freed_chunk(unsigned int * ptr)
{
    printf("[ prev - 0x%08x ][ size - 0x%08x ][ fd - 0x%08x ][ bk -  
0x%08x ] - Chunk 0x%08x - Freed\n", \
        *(ptr-2),
        *(ptr-1),
        *ptr,
        *(ptr+1),
        (unsigned int)(ptr-2));
}
```

```
int main()
{
    unsigned int * ptr[LEN];
    unsigned int lengths[] = {32, 32, 32, 32, 32}; int i;

    printf("mallocing...\n");
    for(i = 0; i < LEN; i++)
        ptr[i] = malloc(lengths[i]);

    for(i = 0; i < LEN; i++)
        print_inuse_chunk(ptr[i]);

    printf("\nfreeing all chunks...\n");
    for(i = 0; i < LEN; i++)
        free(ptr[i]);

    for(i = 0; i < LEN; i++)
        print_freed_chunk(ptr[i]);

    return 0;}
```

# Heap Overflow

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect
- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap

# code/heapoverflow1

```
void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; print_flag() at %p\n", fly,
    print_flag);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```

# code/heapoverflow1

```
void fly()
{
    printf("Flying ...\\n");
}
```

```
typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; print_flag() at %p\\n", fly,
    print_flag);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\\n", p1);

    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```

Airplane 2

Airplane 1



# code/heapoverflow1

```
void secret()
{
    printf("The secret is bla bla...\n");
}

void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; secret() at %p\n", fly, secret);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

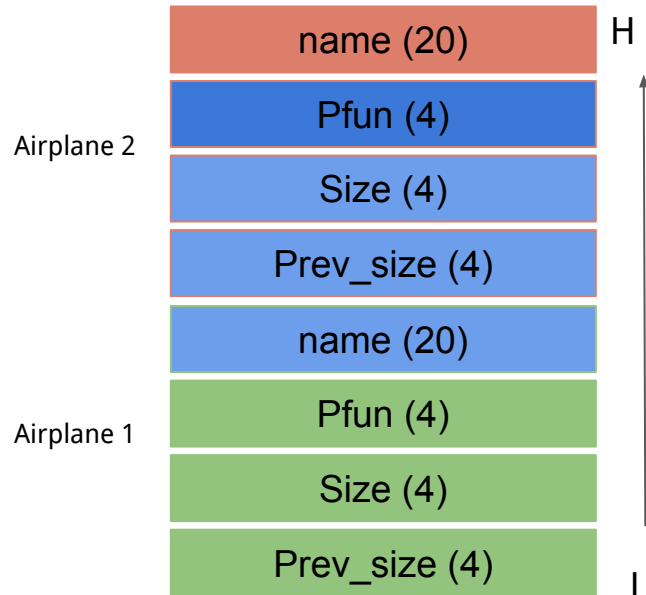
    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```



Exploit looks like

```
python -c "print 'a\n' + 'a'*28 + '\x4d\x62\x55\x56'" | ./heapoverflow32
```

## Use after free (UAF)

A class of vulnerability where data on the heap is freed, but a leftover reference or 'dangling pointer' is used by the code as if the data were still valid.

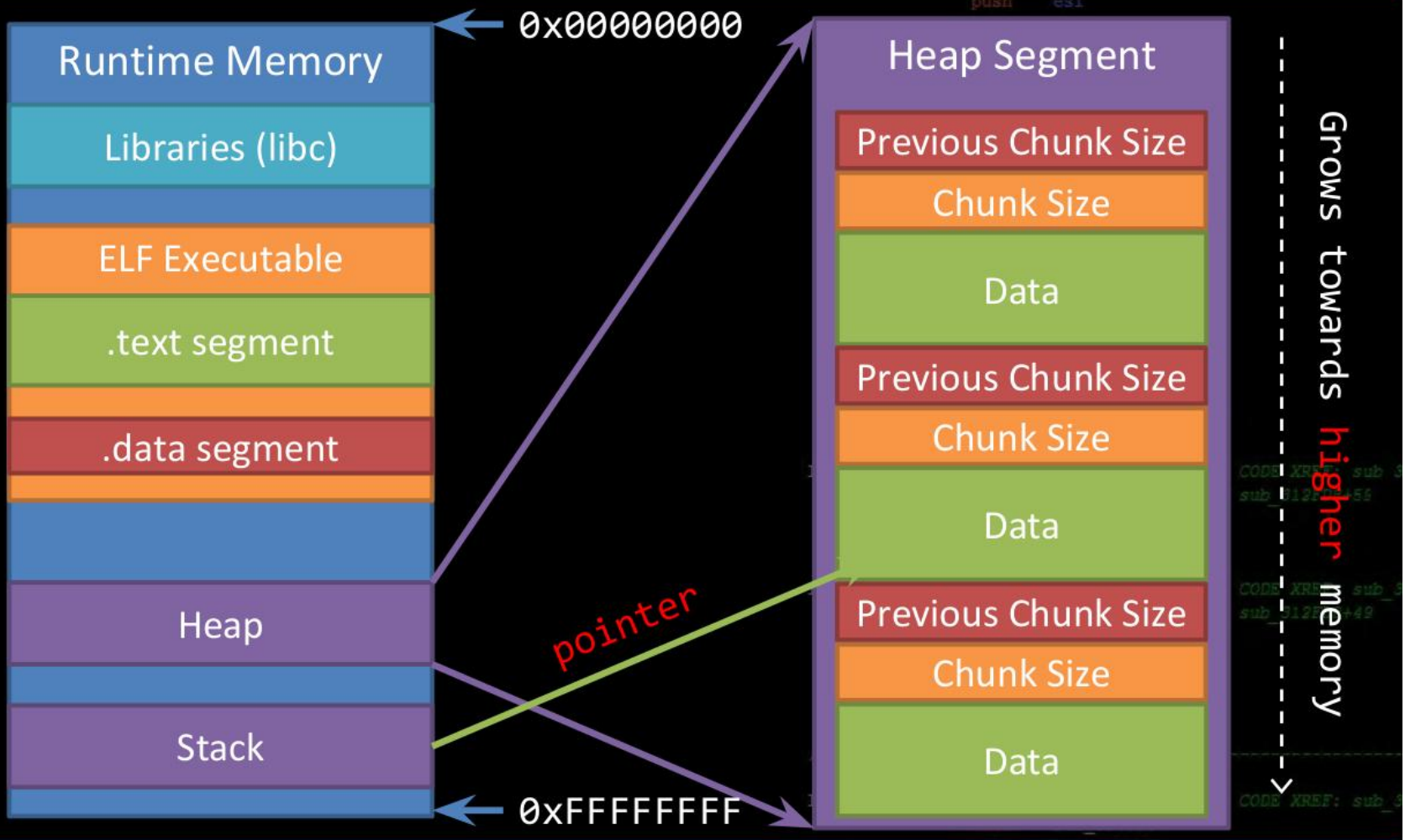
Most popular in Web Browsers, complex programs

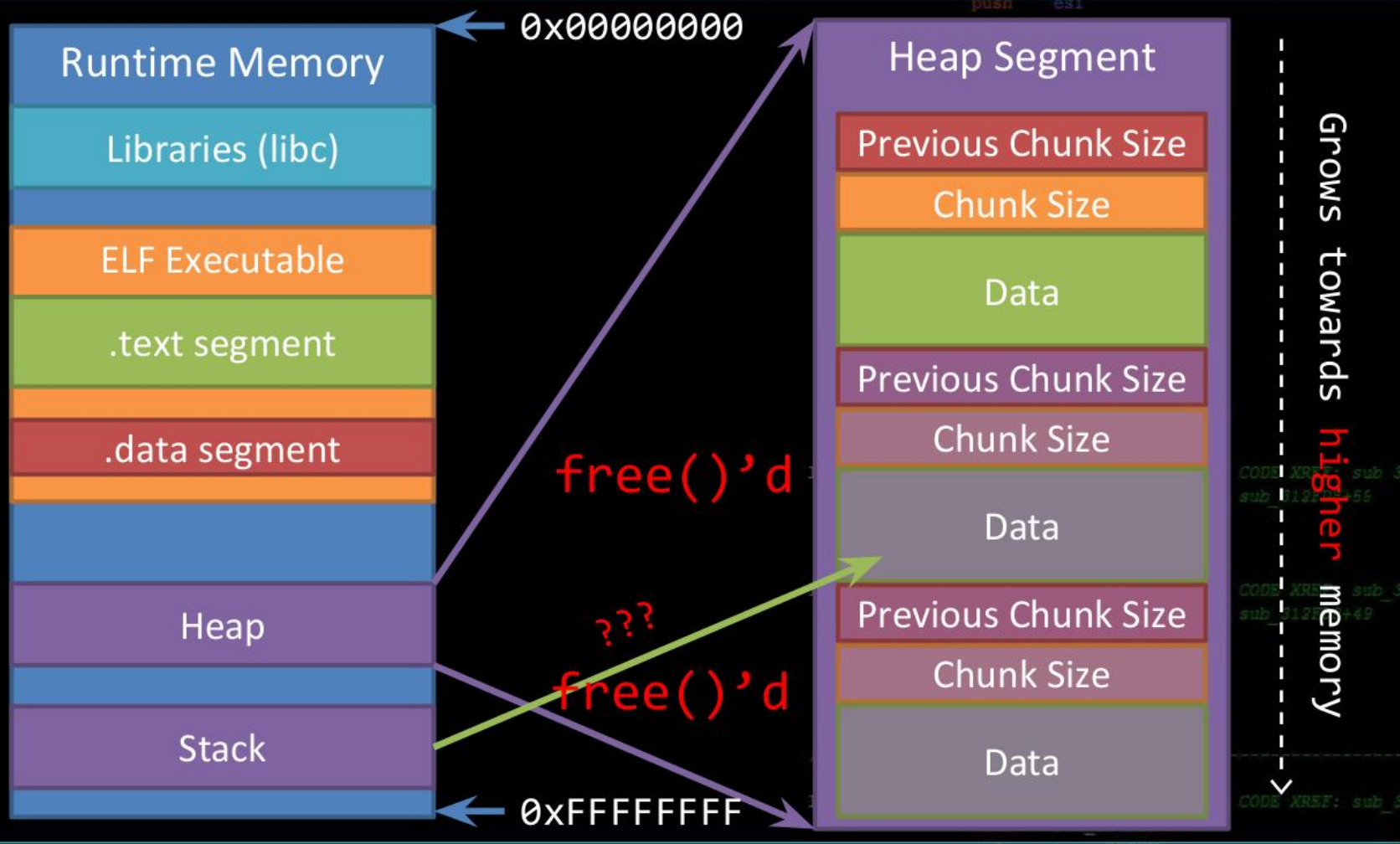


## The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	<a href="#">CWE-787</a>	Out-of-bounds Write	64.20	62	0
2	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	<a href="#">CWE-20</a>	Improper Input Validation	20.63	20	0
5	<a href="#">CWE-125</a>	Out-of-bounds Read	17.67	1	-2 ▼
6	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	<a href="#">CWE-416</a>	Use After Free	15.50	28	0
8	<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	<a href="#">CWE-476</a>	NULL Pointer Dereference	7.15	0	+4 ▲
12	<a href="#">CWE-502</a>	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	<a href="#">CWE-190</a>	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	<a href="#">CWE-287</a>	Improper Authentication	6.35	4	0
15	<a href="#">CWE-798</a>	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	<a href="#">CWE-862</a>	Missing Authorization	5.53	1	+2 ▲
17	<a href="#">CWE-77</a>	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	<a href="#">CWE-306</a>	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	<a href="#">CWE-119</a>	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	<a href="#">CWE-276</a>	Incorrect Default Permissions	4.84	0	-1 ▼
21	<a href="#">CWE-918</a>	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	<a href="#">CWE-362</a>	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	<a href="#">CWE-400</a>	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	<a href="#">CWE-611</a>	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	<a href="#">CWE-94</a>	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

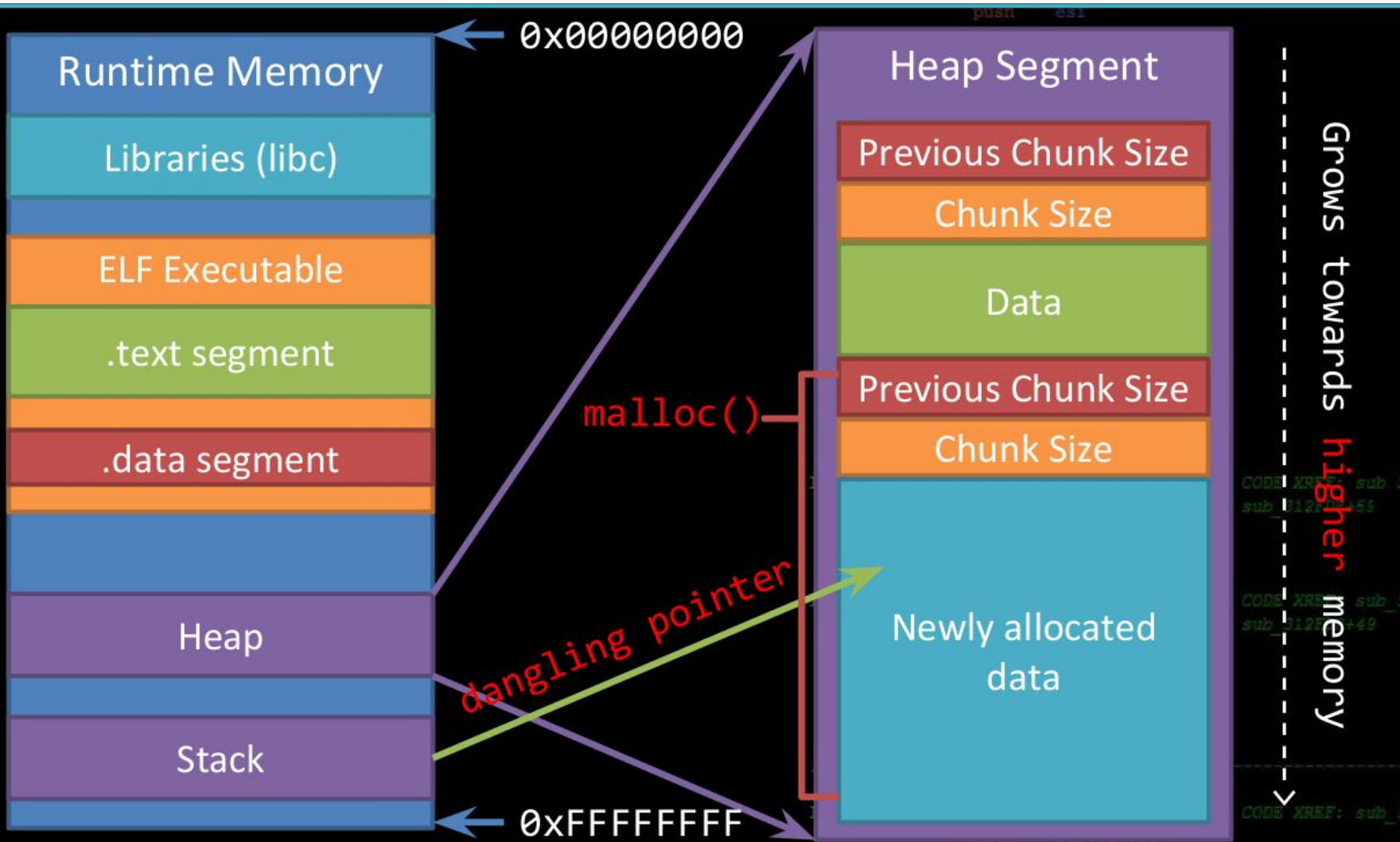




# Dangling Pointer

## Dangling Pointer

- A left over pointer in your code that references free'd data and is prone to be re-used
- As the memory it's pointing at was freed, there's no guarantees on what data is there now
- Also known as stale pointer, wild pointer



# Exploit UAF

To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

# code/heapoverflow2

```
void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;

typedef struct car
{
    int volume;
    char name[20];
} car;
```

```
int main()
{ printf("fly() at %p; print_flag() at %u\n", fly, (unsigned int)print_flag);

    struct airplane *p = malloc(sizeof(airplane));
    printf("Airplane is at %p\n", p);
    p->pfun = fly;
    p->pfun();
    free(p);

    struct car *p1 = malloc(sizeof(car));
    printf("Car is at %p\n", p1);

    int volume;
    printf("What is the volume of the car?\n");
    scanf("%u", &volume);
    p1->volume = volume;

    p->pfun();
    free(p);
    return 0;
}
```

# Dmalloc (using glibc 2.3 as an example)

```
struct malloc_chunk
{
    INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T size; /* Size in bytes, including overhead. */

    struct malloc_chunk* fd; /* double links -- used only if free. */
    struct malloc_chunk* bk;

};

typedef struct malloc_chunk* mchunkptr;
```

**Mem** is the pointer returned by malloc() call, while **chunk pointer** is what malloc considers the start of the chunk.

The whole heap is bounded from top by a **wilderness** chunk. In the beginning, this is the only chunk existing and malloc first makes allocated chunks by splitting the wilderness chunk.

glibc 2.3 allows for many heaps arranged into several **arenas**—one arena for each thread

– From the book “Buffer Overflow Attacks: Detect, Exploit, Prevent” Syngree



# Consolidating chunks when free()-d

When a previously allocated chunk is free()-d, it can be either consolidated with previous (backward consolidation) and/or follow (forward consolidation) chunks, if they are free.

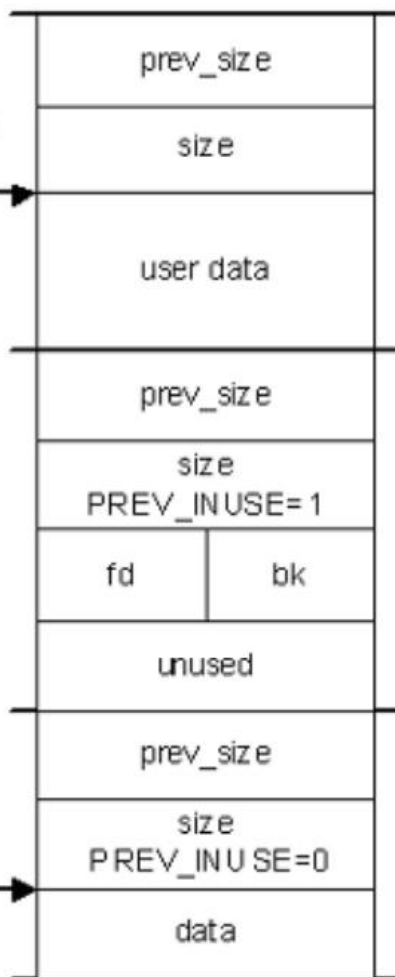
This ensures that there are no two adjacent free chunks in memory. The resulting chunk is then placed in a **bin**, which is a **doubly linked list of free chunks of a certain size**.

There is a set of bins for chunks of different sizes:

- 64 bins of size 8 ■ 32 bins of size 64 ■ 16 bins of size 512
- 8 bins of size 4096 ■ 4 bins of size 32768 ■ 2 bins of size 262144
- 1 bin of size what's left

chunk A,  
being freed

A →



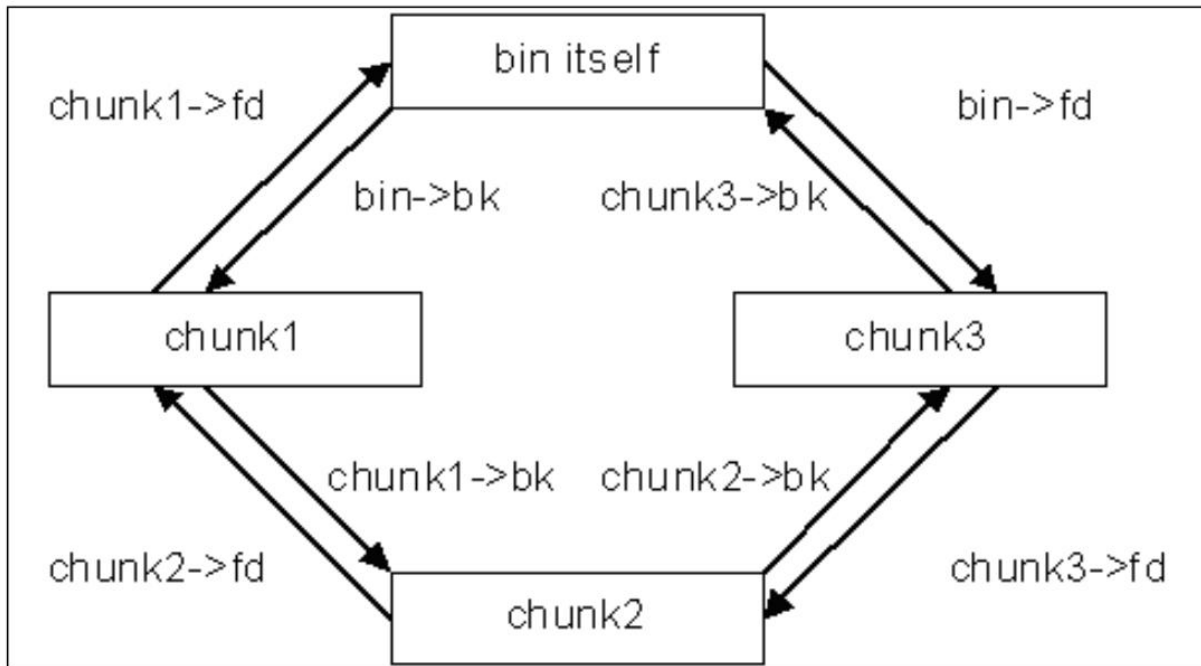
chunk B,  
free

chunk C,  
allocated

C →

chunk A will be  
forward consolidated  
with B

# Example Bin with Three Free Chunks



FD and BK are pointers to “next” and “previous” chunks inside a linked list of a bin, ***not adjacent physical chunks***.

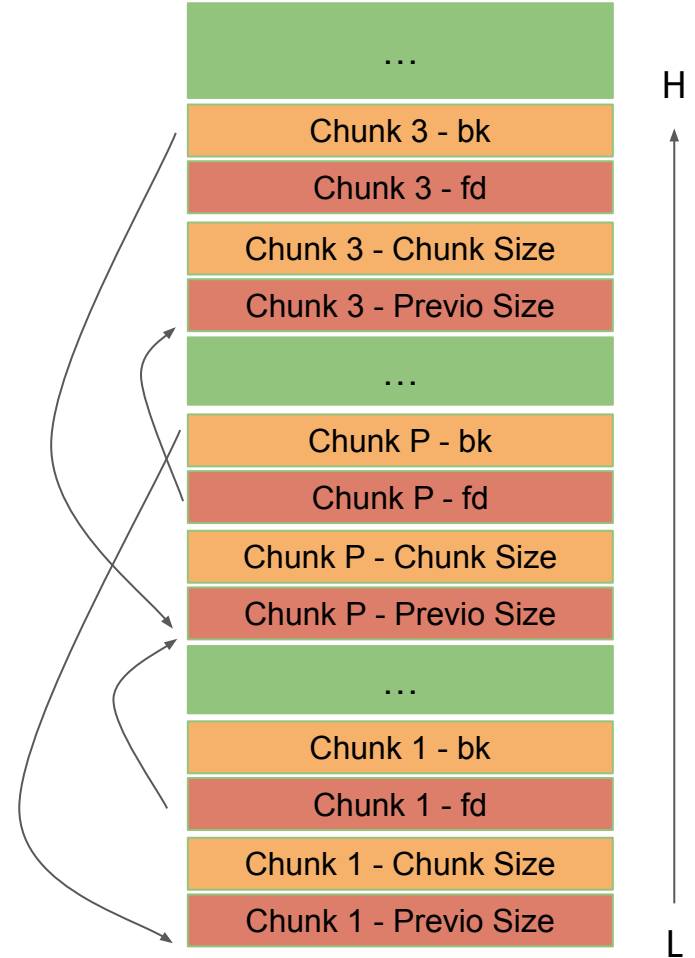
Pointers to chunks, physically next to and previous to this one in memory, can be obtained from current chunks by using **size** and **prev\_size** offsets.

# Pointers to physically next to and previous chunk

```
/* Ptr to next physical malloc_chunk. */  
#define next_chunk(p) ((mchunkptr)((char*)(p) + ((p)->size & ~PREV_INUSE) ))  
  
/* Ptr to previous physical malloc_chunk */  
#define prev_chunk(p) ((mchunkptr)((char*)(p) - ((p)->prev_size) ))
```

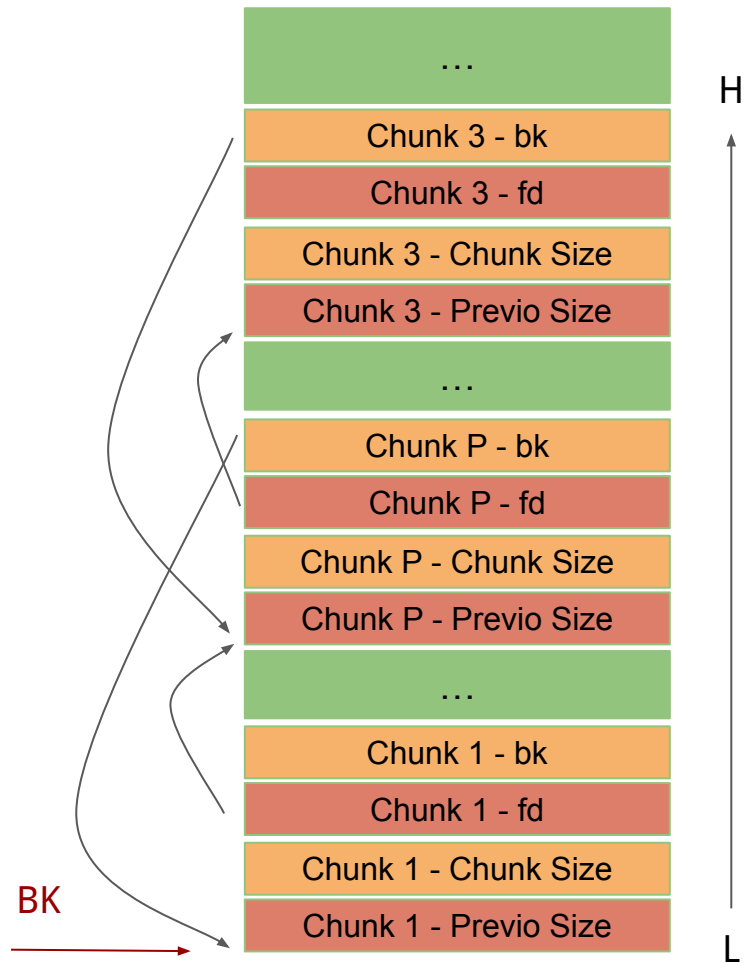
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



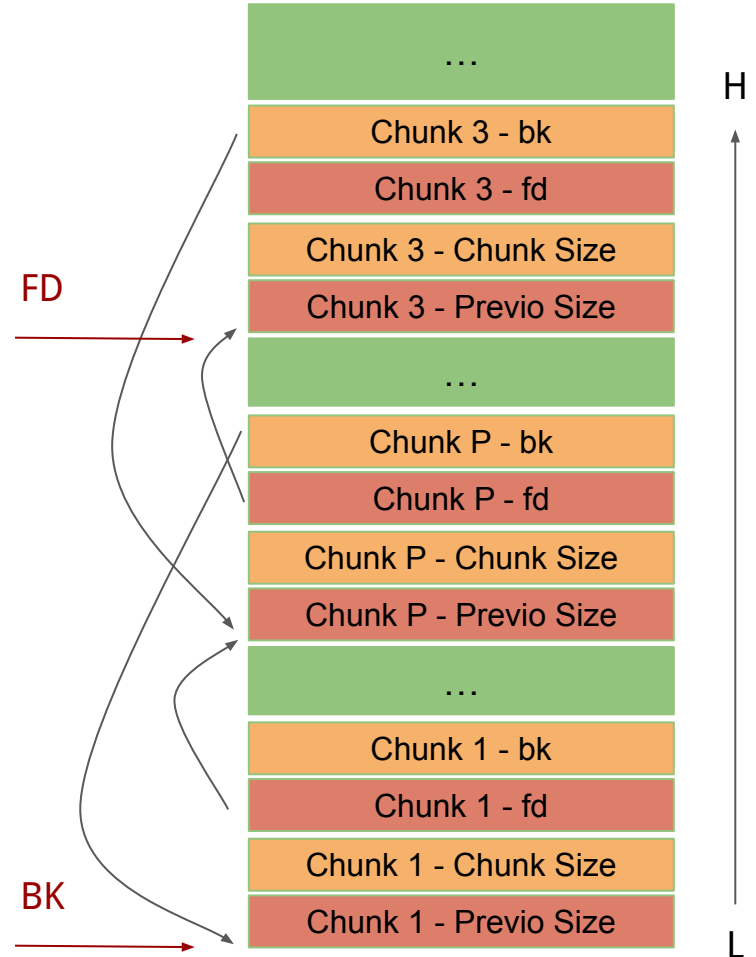
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



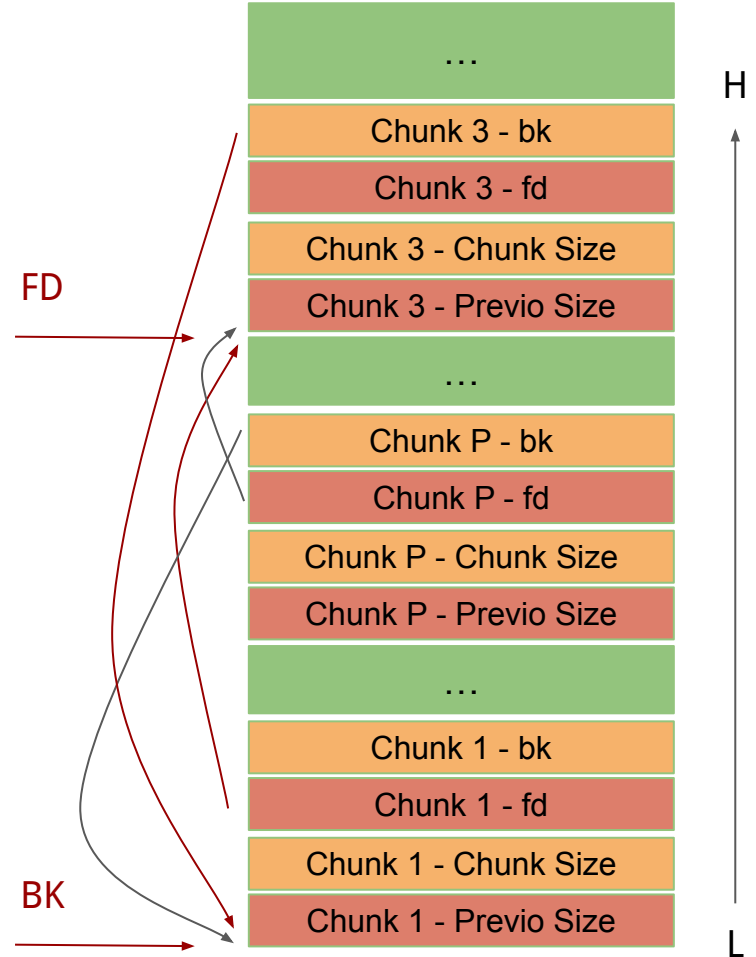
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```





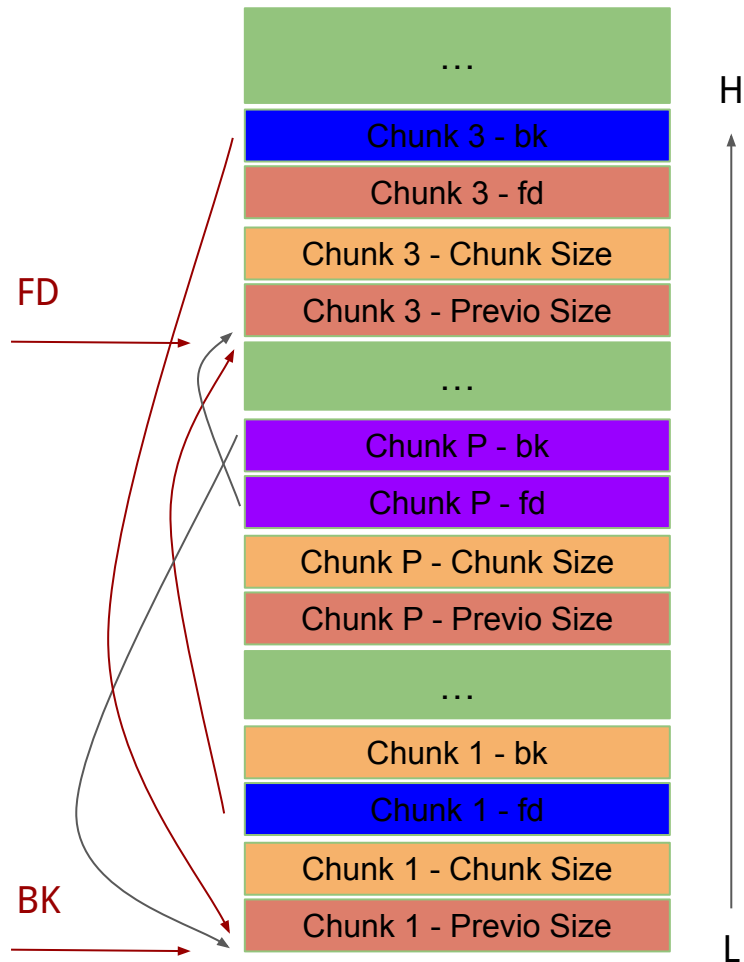
# Unlink() from an Attacker's Point of View

```
*(P->fd+12) = P->bk;  
// 4 bytes for size, 4 bytes for prev_size and 4 bytes for fd
```

```
*(P->bk+8) = P->fd;  
// 4 bytes for size, 4 bytes for prev_size
```

Arbitrary write attack?

If an attacker is able to overwrite these two pointers and force the call to unlink(), he can overwrite any memory location.



# The free() Algorithm

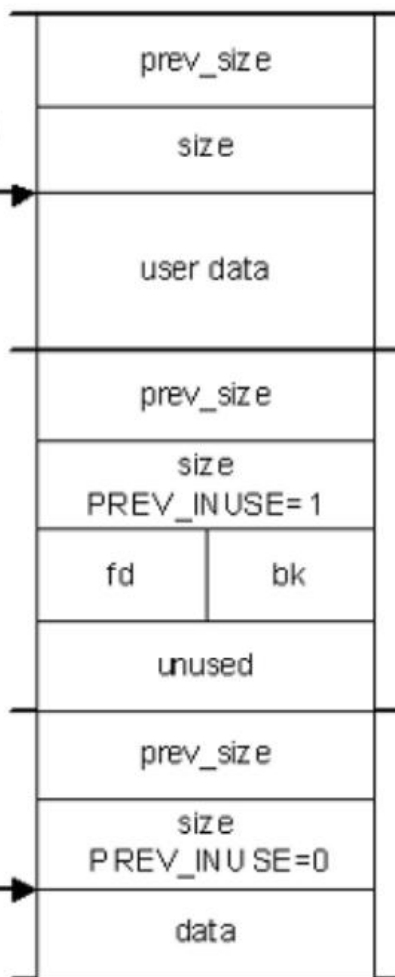
- `free(0)` has no effect.
- If the chunk was allocated via `mmap`, it is released via `munmap()`. Only large chunks are MMAP-ped, and we are not interested in these.
- If a returned chunk borders the current high end of memory (wilderness chunk), it is consolidated into the wilderness chunk, and if the total unused topmost memory exceeds the trim threshold, `malloc_trim()` is called.
- Other chunks are consolidated as they arrive, and placed in corresponding bins.

# The free() Algorithm - last case

- If no adjacent chunks are free, then the freed chunk is simply linked into corresponding with bin via frontlink().
- If the chunk next in memory to the freed one is free and if this next chunk borders on wilderness, then both are consolidated with the wilderness chunk.
- If not, and the previous or next chunk in memory is free and they are not part of a most recently split chunk (this splitting is part of malloc() behavior and is not significant to us here), they are taken off their bins via unlink(). Then they are merged (through forward or backward consolidation) with the chunk being freed and placed into a new bin according to the resulting size using frontlink(). If any of them are part of the most recently split chunk, they are merged with this chunk and kept out of bins. This last bit is used to make certain operations faster.

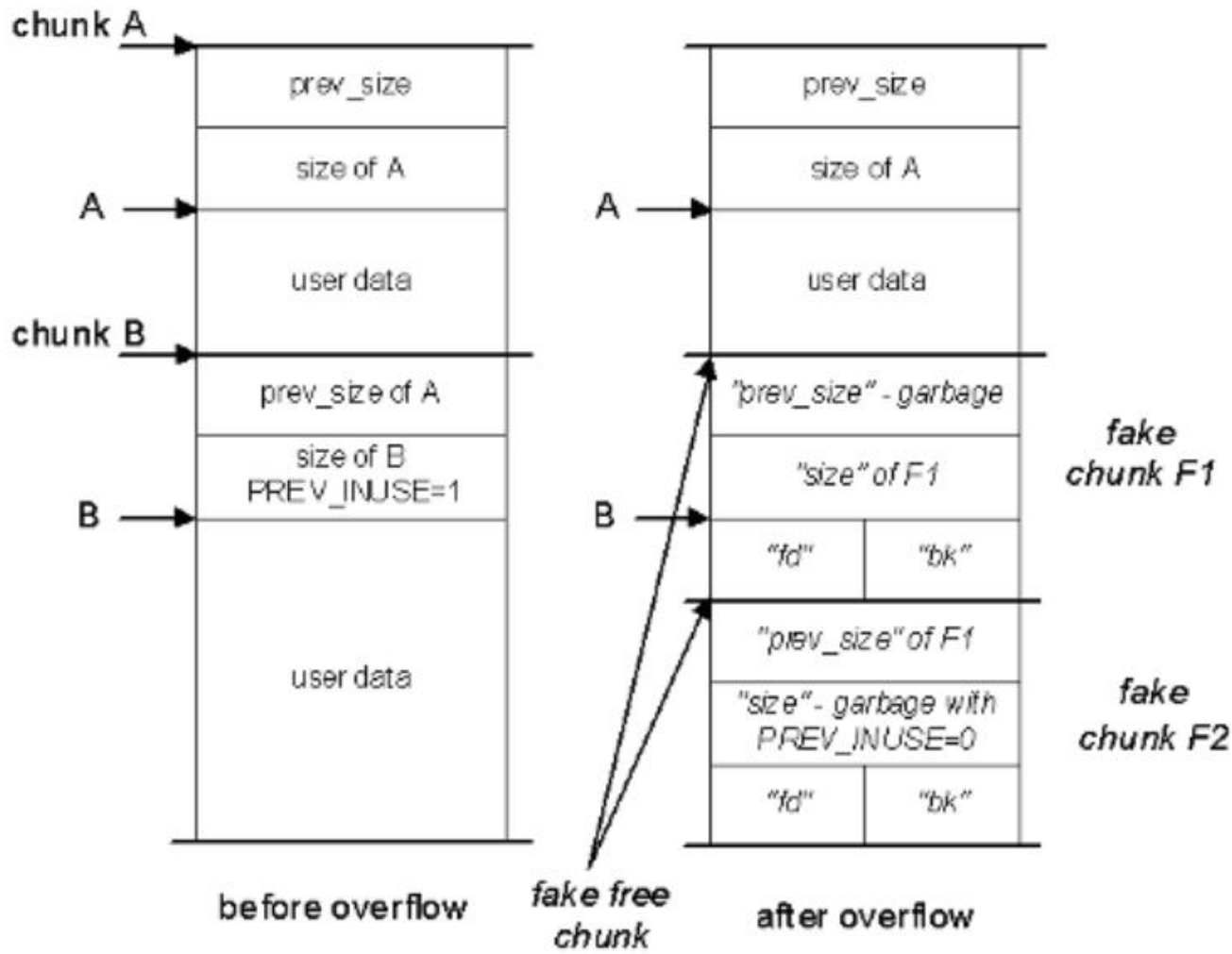
chunk A,  
being freed

A →



chunk A will be  
forward consolidated  
with B

# lower addresses



1. Overwrite A and B
2. Create a fake chunk F1 and F2, so that when free(A), unlink(F1) is also called.
3. F1->FD has the address we want to overwrite and F1->BK has the data we want to overwrite