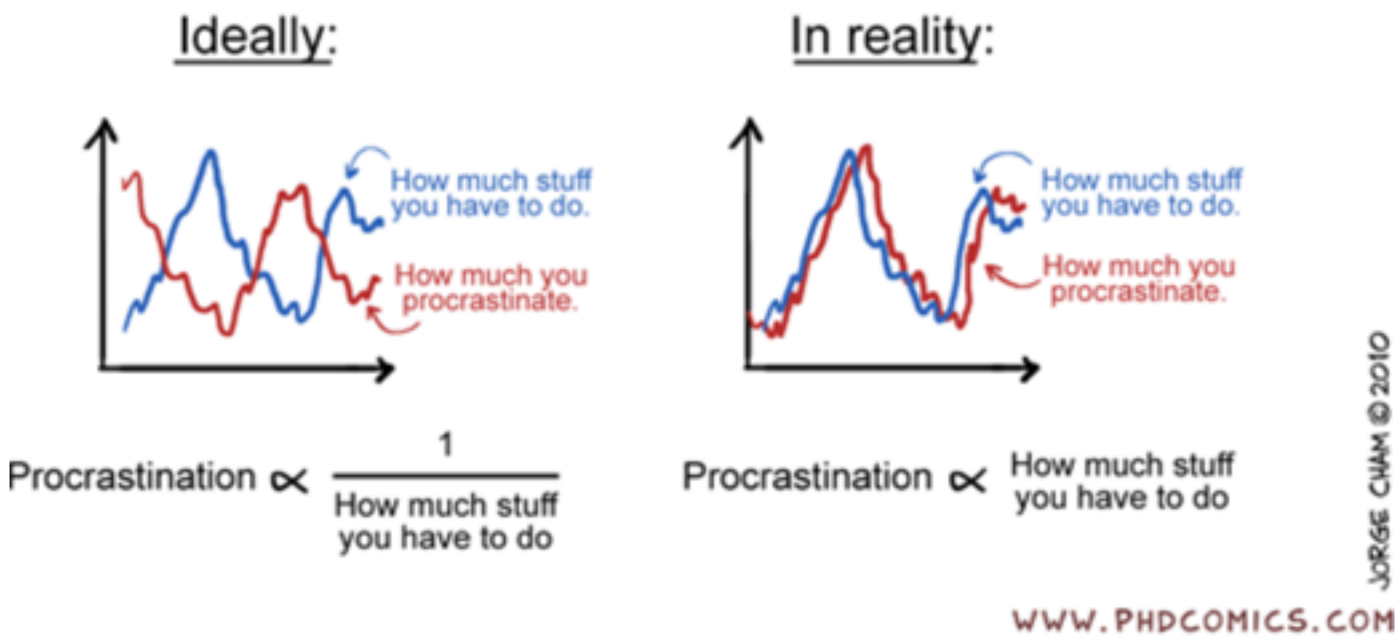
# Heap Overflows

Secure Programming
Lecture 13

# Announcement

- Homework 2 has been graded
  - Let me know of any problems, issues, things that are unclear
- Homework 3 deadline is coming up **Tuesday March 11, 11:59pm**
  - Don't procrastinate




title: "Procrascorrelation" - originally published 10/27/2010

# In the news

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
     goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
     goto fail;
     goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
     goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

hlps://www.imperialviolet.org/2014/02/22/applebug.html

# Heap basics

- Region for dynamically allocated memory
  - Dynamic allocation
  - Dynamic de-allocation (freeing)
- Operating system
  - `sbrk()`: allows to change the “program break”, i.e., the end of the process’ data segment
  - `mmap()`
- Heap manager (“allocator”)
  - Handles memory requests from user programs
  - Acts as interface to the `brk()` system call
  - Standard `malloc()`, `free()`, `realloc()`, `calloc()` functions
  - Different implementation in different systems

# Heap management implementations

| Algorithm | Operating system |
|---|---|
| ***Doug Lea's ptmalloc*** | ***Linux glibc*** |
| System V | Solaris |
| ***BSD phk*** | ****BSD*** |
| RtlHeap | Windows |

# Heap basic

```
#define MAX_PATH_LEN 1024

void handle_request(struct req *r) {
  char *s = (char *) malloc(MAX_PATH_LEN);
  strncpy(s, r->path, MAX_PATH_LEN – 1);
  s[MAX_PATH_LEN  - 1] = ‘\0’;

  /* do something with s */

  free(s);
}
```

# Heap overflows

```
#define MAX_PATH_LEN 1024

void handle_request(struct req *r) {
  char *s = (char *) malloc(MAX_PATH_LEN);
  strcpy(s, r->path);
  s[MAX_PATH_LEN  - 1] = ‘\0’;

  /* do something with s */

  free(s);
}
```

# Naive malloc

- Let's try to think how we could implement a memory allocator

# Common design and requirements

- Common goals
  - Efficiency (time): malloc, free, realloc should be fast
  - Efficiency (space): don’t waste space
- Book-keeping information (metadata):
  - Pointers, size values, indexes to regions
  - Commonly done in-band, i.e., along-side the actual data values (for space efficiency)
- Coalescing of unused regions
  - Merge two or more free pieces of memory that are next to each other (avoid fragmentation)

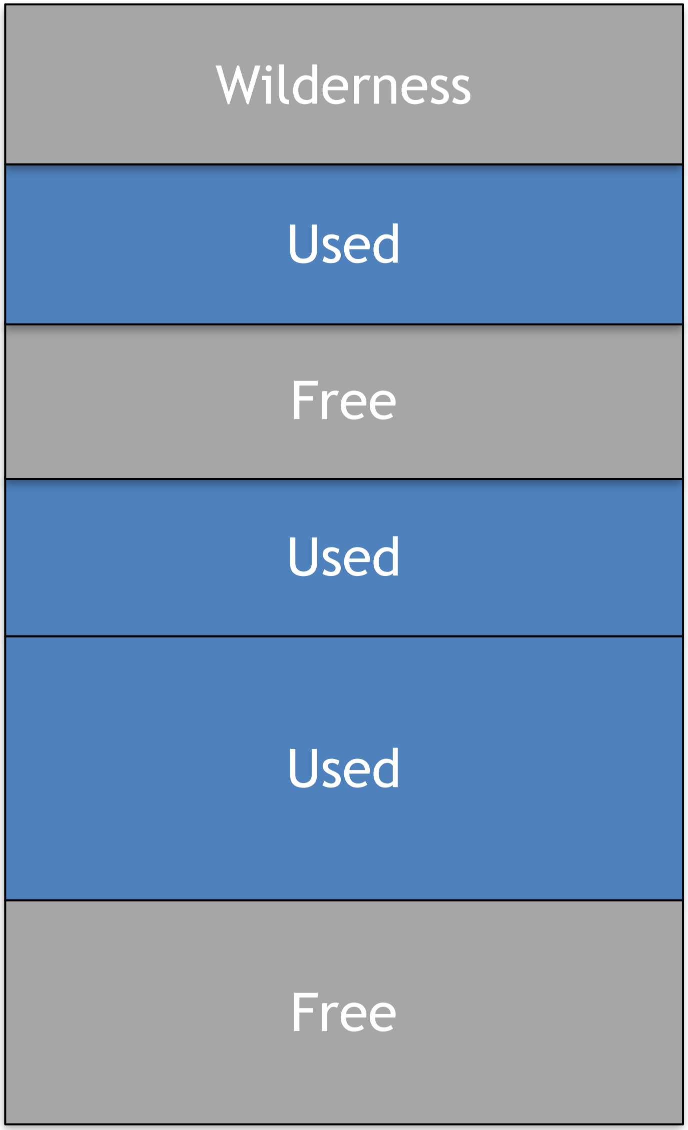
# Heap overflow vulnerabilities

- Overflow of data allocated on the heap
  - Allocate some memory on the heap (buffer, structure, etc.)
  - Write into this region exceeding its boundaries
- Successful exploitation of heap overflows was demonstrated by Solar Designer (2000)
- Standard reference
  MaXX, Vudo malloc tricks, 2001
- General idea of such attacks: take advantage of the mixing of data and control information on the heap to overwrite critical control structure

# Doug Lea’s malloc

- Memory allocator in glibc
  - Description: http://g.oswego.edu/dl/html/malloc.html
  - Source: http://code.metager.de/source/xref/glibc/malloc/
- Key functionalities
  - Chunk management
  - Bin management
  - Memory allocation
  - Memory deallocation
  - List handling

# Heap memory layout




- Heap is divided into contiguous *chunks* of memory
  - Chunks can be allocated, freed, split, combined
- No two free chunks may be physically adjacent
  - Except “fastbins” (bins containing small chunks < 64 bytes)
- Coalescing via boundary tags design (with space optimizations)
- Best-fit via binning

# malloc_chunk

```
struct malloc_chunk {
  size_t prev_size;
  size_t size;
  struct malloc_chunk *fd;
  struct malloc_chunk *bk
}
```

- “Misleading declaration”
- Use it as a “view” into memory to access fields at known offsets
- Fields have different interpretation depending on this and previous chunk state (allocated or not)

# malloc_chunk

```
struct malloc_chunk {
  size_t prev_size;
  size_t size;
  struct malloc_chunk *fd;
  struct malloc_chunk *bk
}
```

If the previous chunk is free, it contains its size.

Otherwise, it holds user data of the previous chunk (reduce waste)

# malloc_chunk

```
struct malloc_chunk {
  size_t prev_size;
  size_t size;
  struct malloc_chunk *fd;
  struct malloc_chunk *bk
}
```

Holds the chunk size in bytes.

size = requested memory
+ 8 bytes (size_t * 2)
- 4 bytes (prev_size of next chunk)

Always multiple of 8 (alignment). Last 3 bits of size are logically 0.

Reuse last bits:
P: PREV_INUSE (0x1)
M: IS_MMAPPED (0x2)

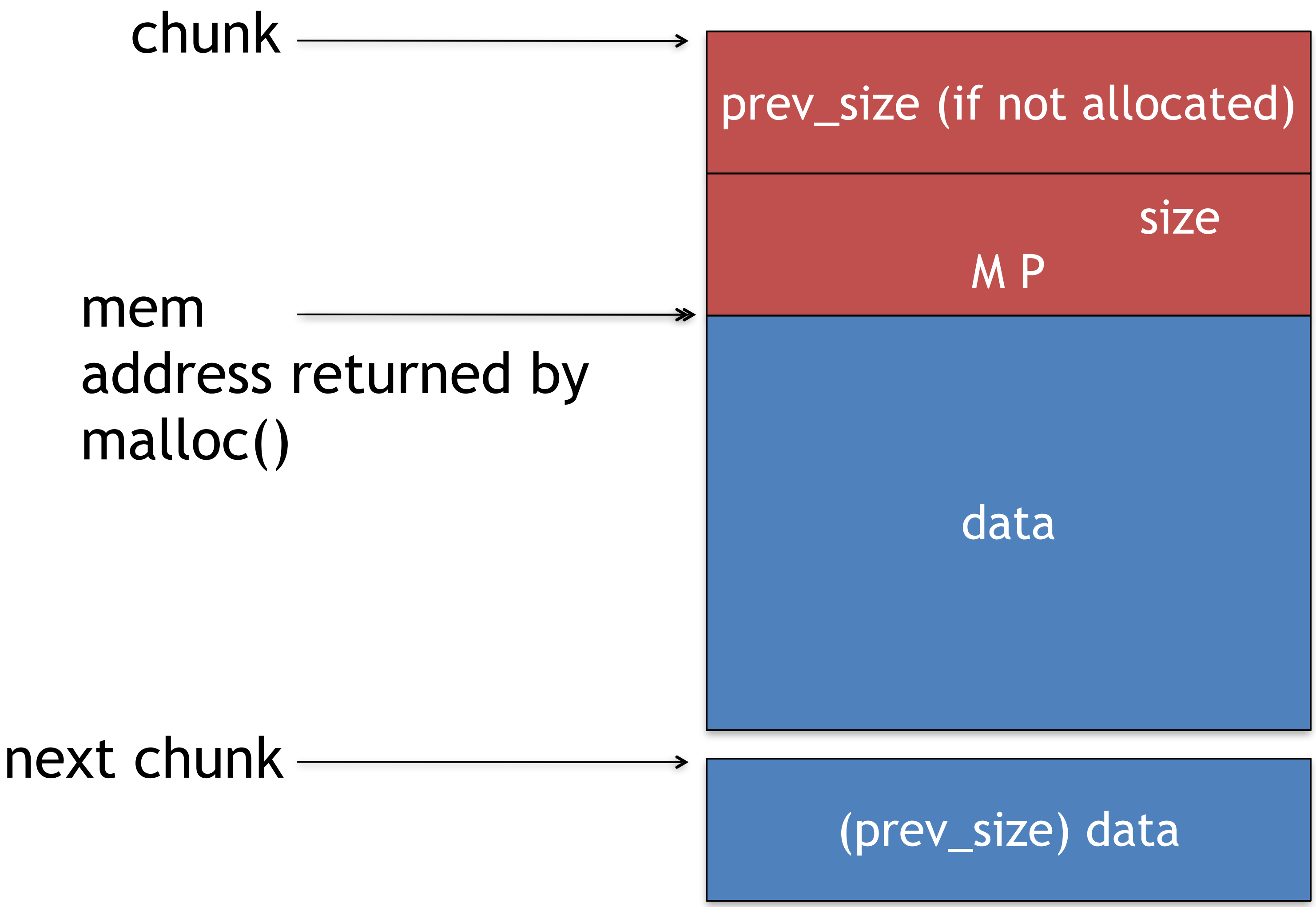
# malloc_chunk

```
struct malloc_chunk {
  size_t prev_size;
  size_t size;
  struct malloc_chunk *fd;
  struct malloc_chunk *bk
}
```

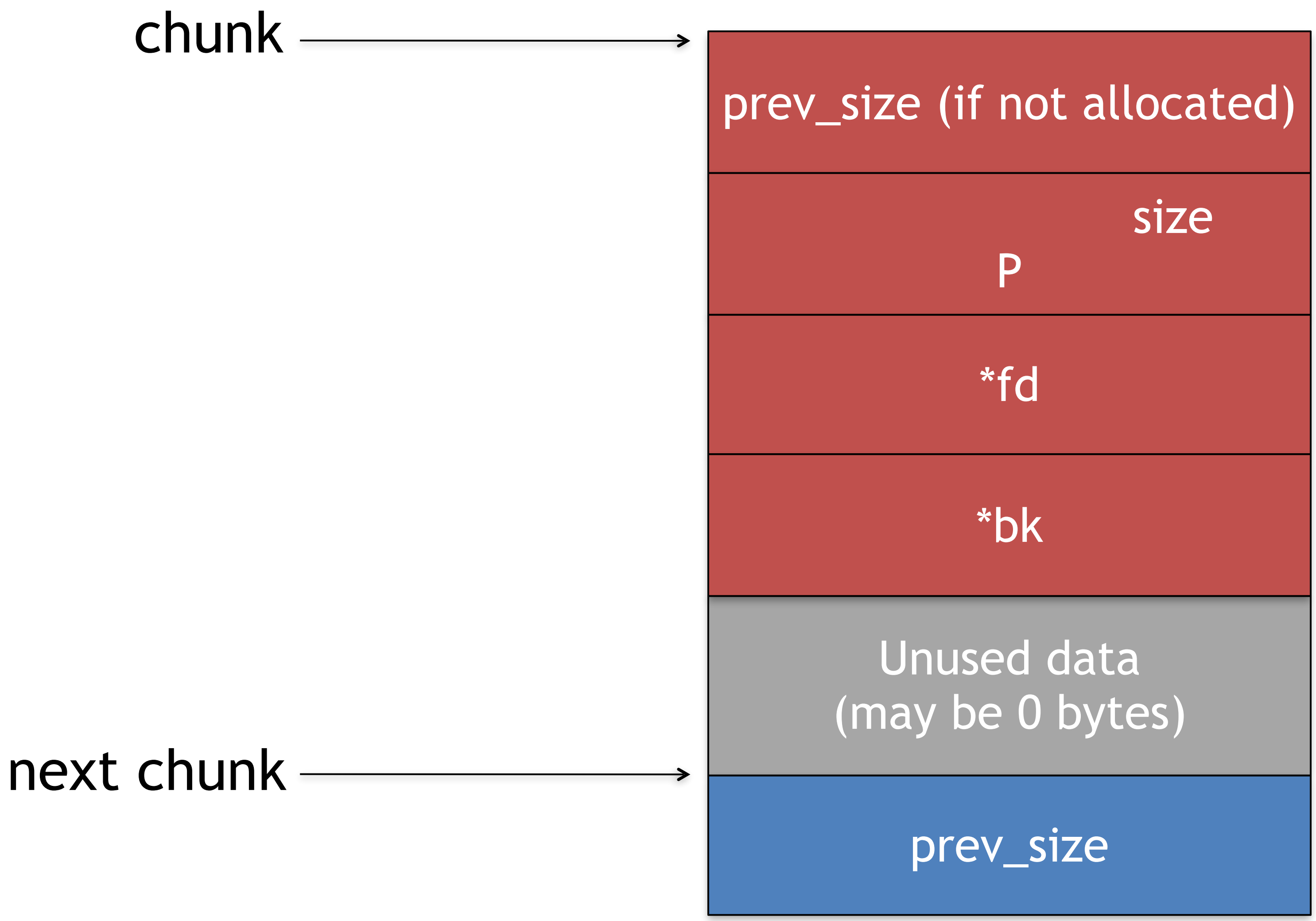
If chunk is free, they contain pointers to next and previous free blocks (doubly-linked list)

If chunk is used, they contain user data.

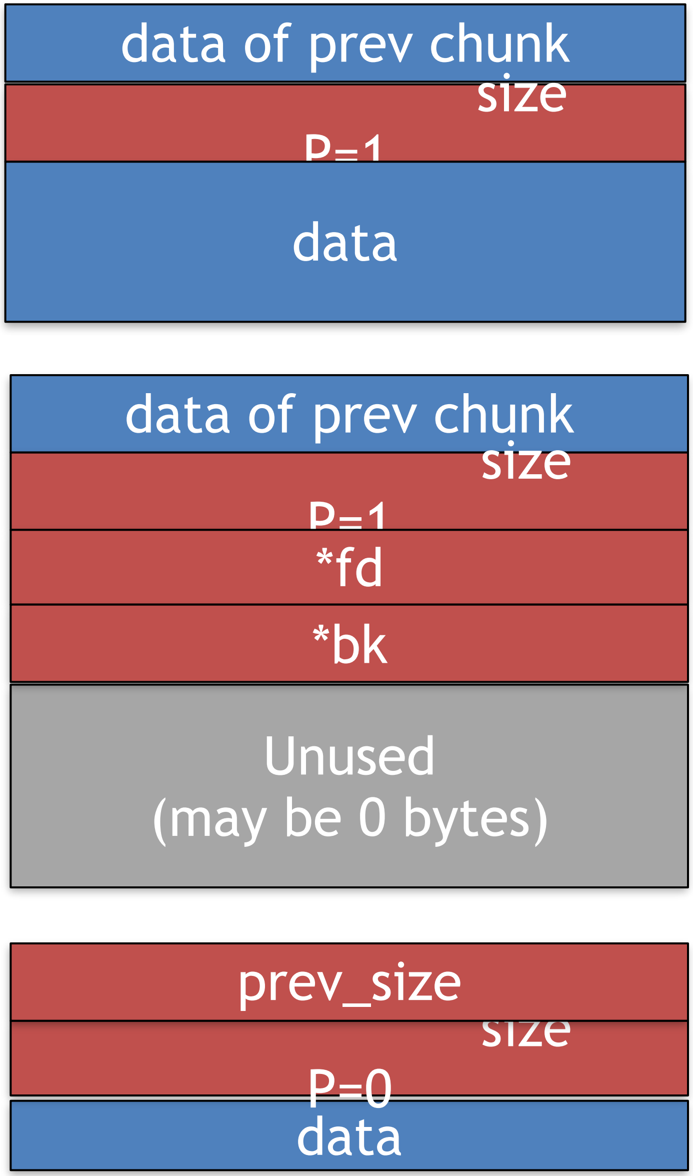
# Allocated chunk



# Free chunk


chunk
prev_size (if not allocated)
size
P
*fd
*bk
Unused data
(may be 0 bytes)
next chunk
prev_size

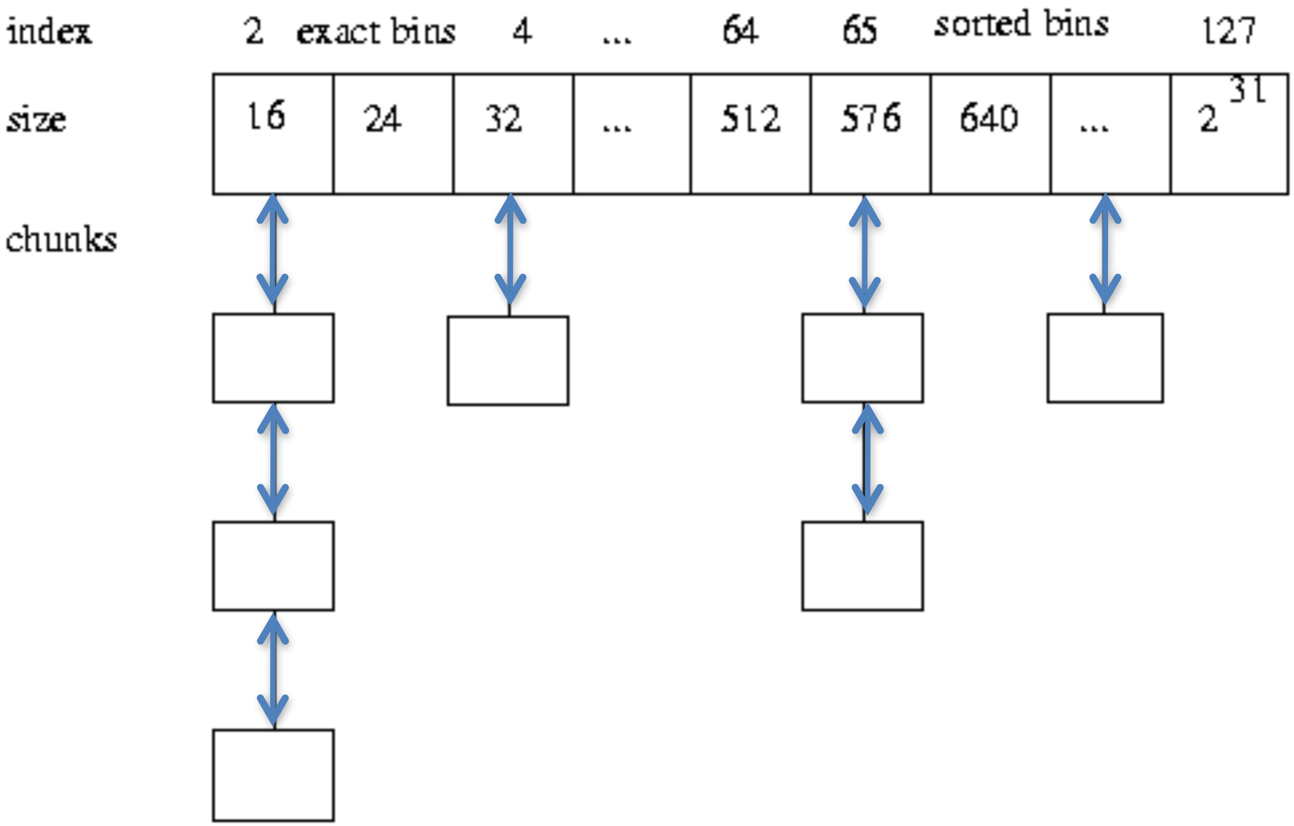
# Chunks



data of prev chunk
size
P=1
data
data of prev chunk
size
P=1
*fd
*bk
Unused
(may be 0 bytes)
prev_size
size
P=0
data

# Bin management

- Available chunks are maintained in bins grouped by size



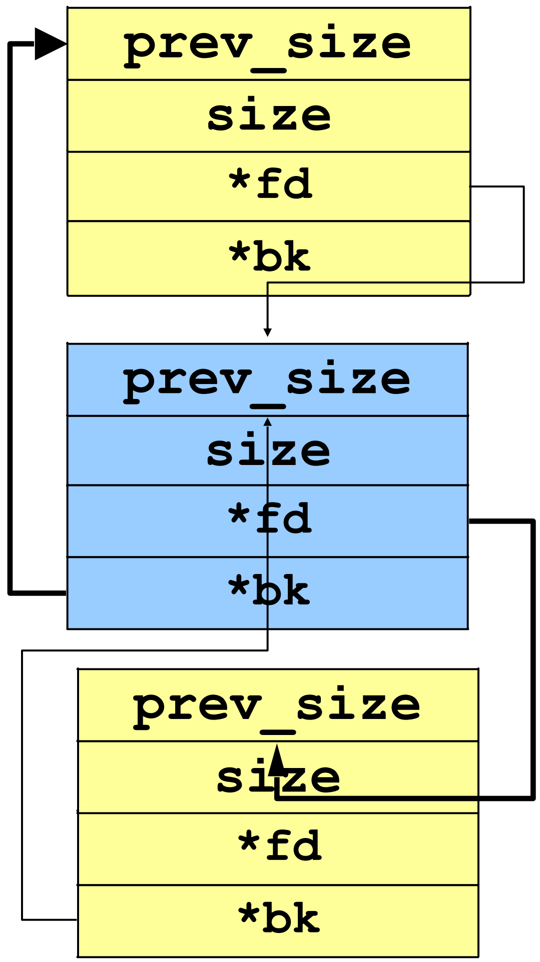
# List handling: `unlink()`

- When chunks are allocated or freed, their entries have to be taken off or inserted into the appropriate list
- Macro unlink is used to take off a chunk with its pointers FD and BK

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

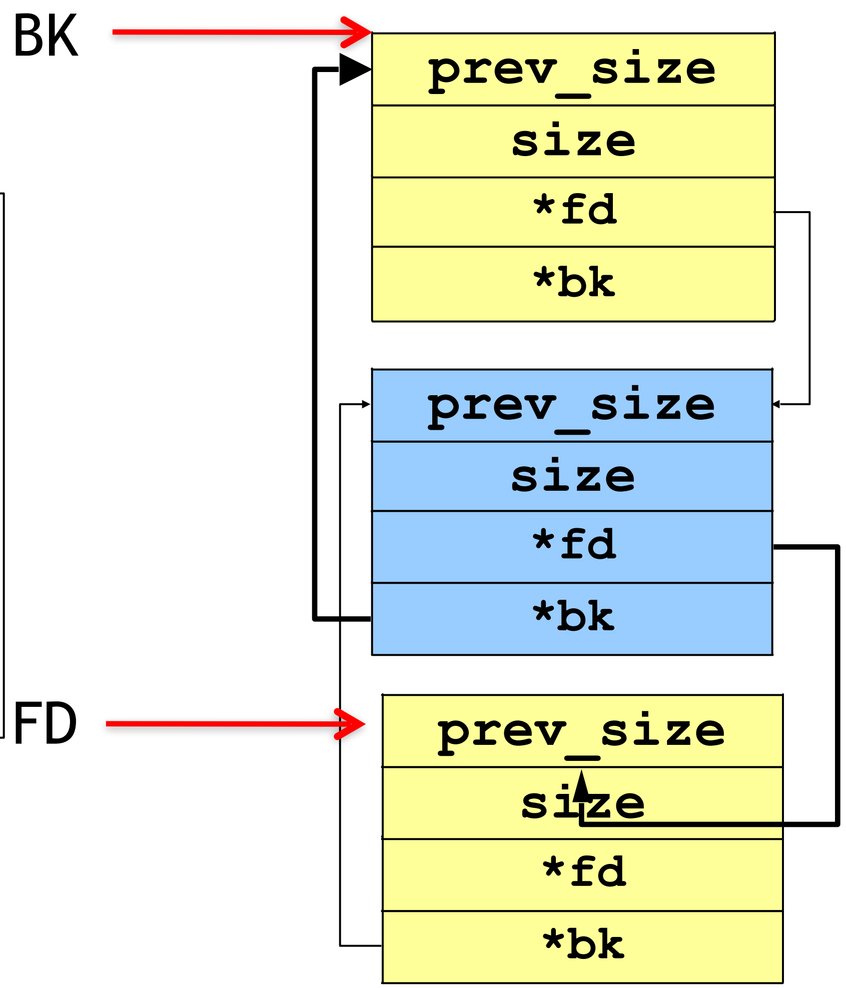
# List handling: unlink()

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



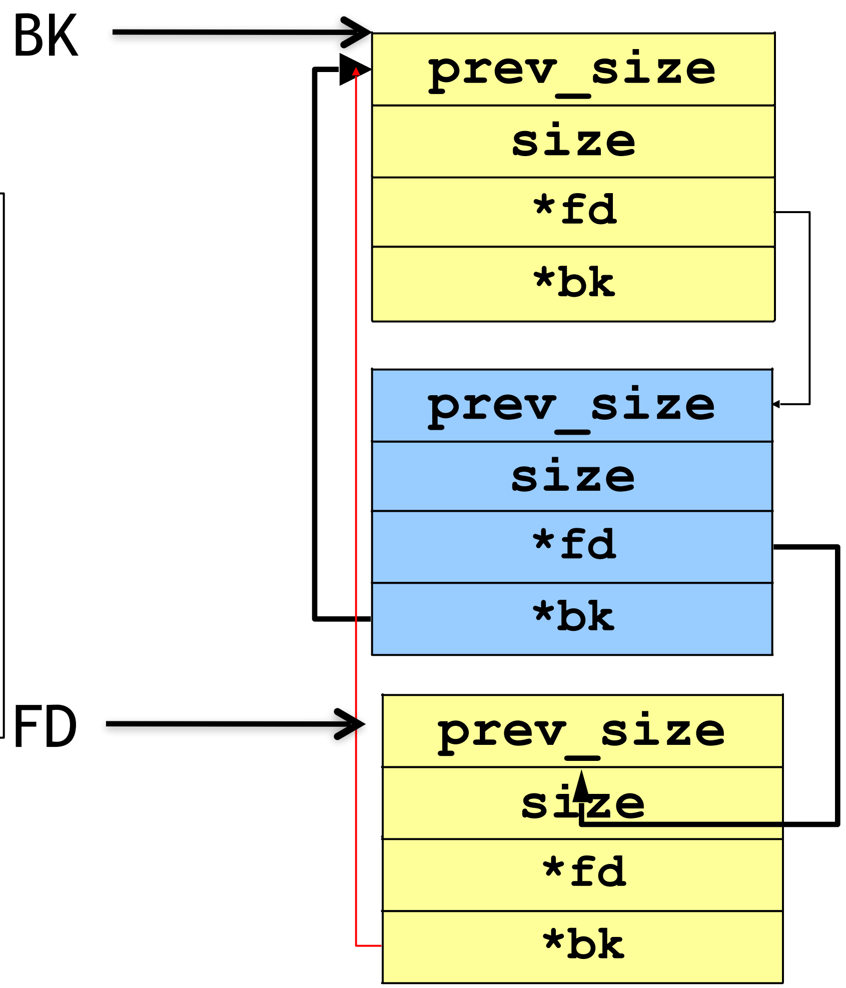
# List handling: unlink()

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



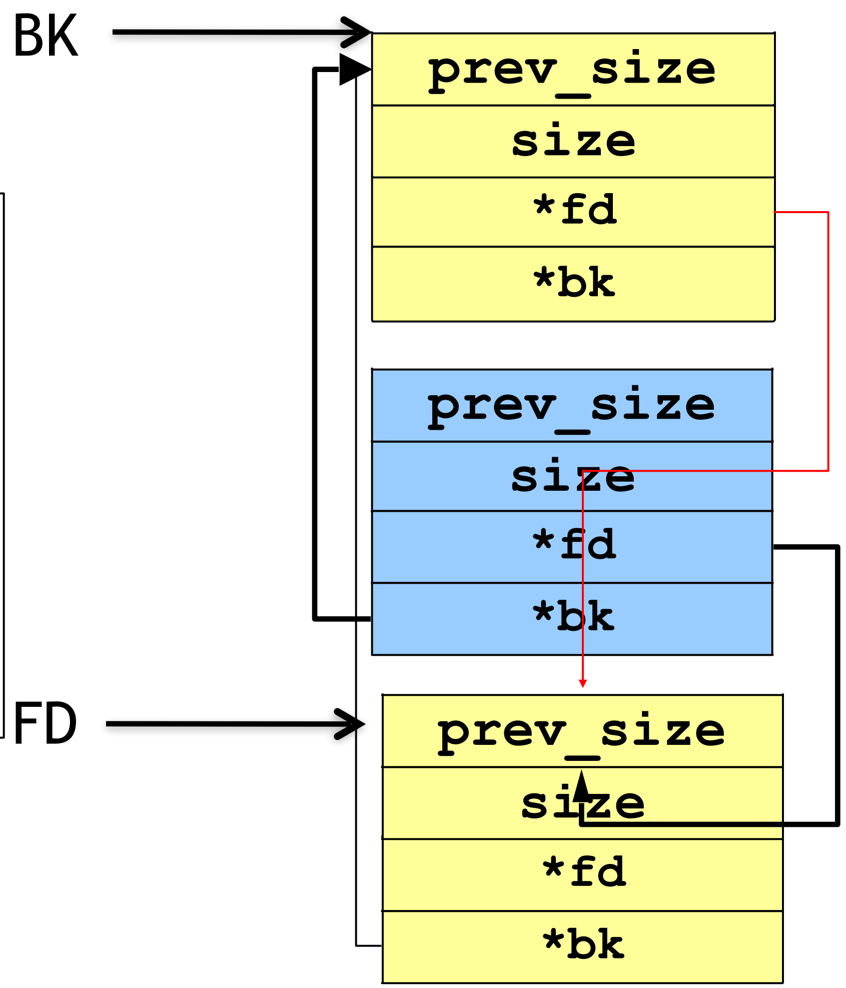
# List handling: unlink()

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



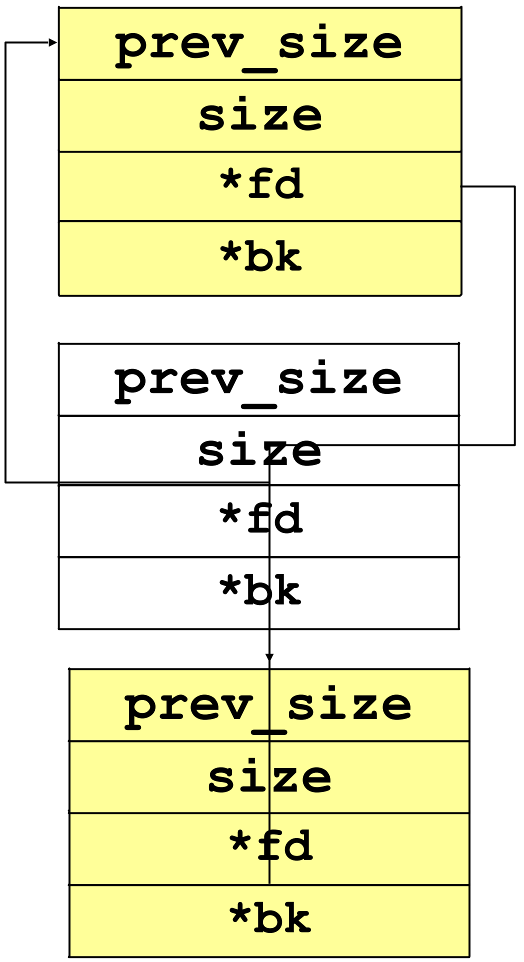
# List handling: unlink()

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



# List handling: unlink()

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



# Exploiting the `unlink()` macro

- What happens if a chunk is overflown?
  - Control data stored in next chunk is overwritten
  - Attacker can set up a “fake chunk” and abuse unlike()

```
P:    struct malloc_chunk {
P+0:    size_t prev_size;
P+4:    size_t size;
P+8:    struct malloc_chunk *fd;
P+c:    struct malloc_chunk *bk
      }
```

```
BK = P->bk;
FD = P->fd;
FD->bk = BK; // *(P->fd+12) = P->bk
BK->fd = FD; // *(P->bk+8) = P->fd
```

Overwrite address stored in FD + 12  with BK

# Exploiting the unlink() macro

Overwrite an arbitrary memory location with an arbitrary value

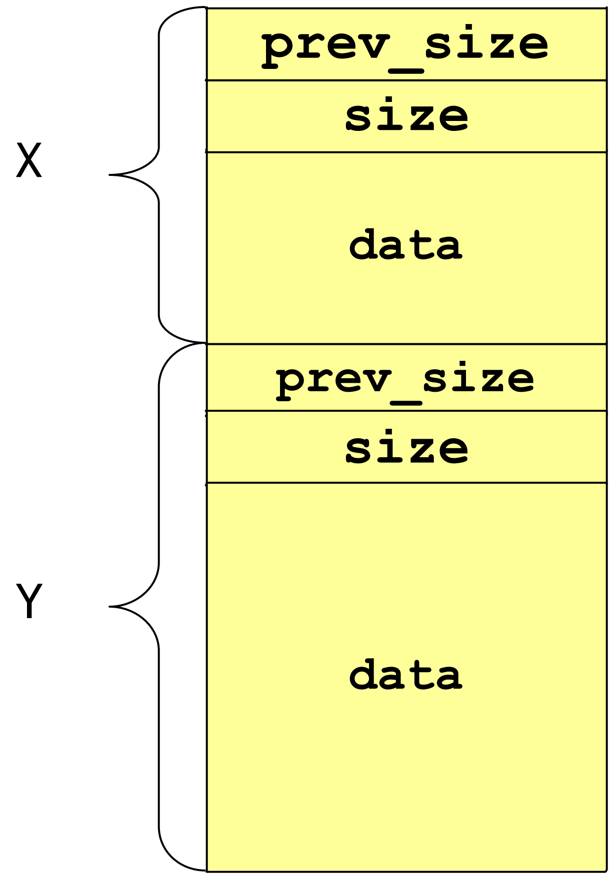
- Overwrite a function pointer with address of the shellcode
  - Often, overwrite an entry in the GOT table

    http://althing.cs.dartmouth.edu/secref/resources/plt-got.txt
    http://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html
- When function is later invoked, shellcode is executed instead

# Exploiting the `unlink()` macro

1. Vulnerable program allocates two adjacent memory chunks, named X and Y
2. The attacker overflows the chunk X creating two fake (free) chunks over Y, called W and Z
3. When X is freed, it will be merged with W and the unlink() macro will be called
4. Since W and Z are under the attacker’s control, arbitrary values can be specified for their headers

# Exploiting the unlink() macro



# Exploiting the unlink() macro

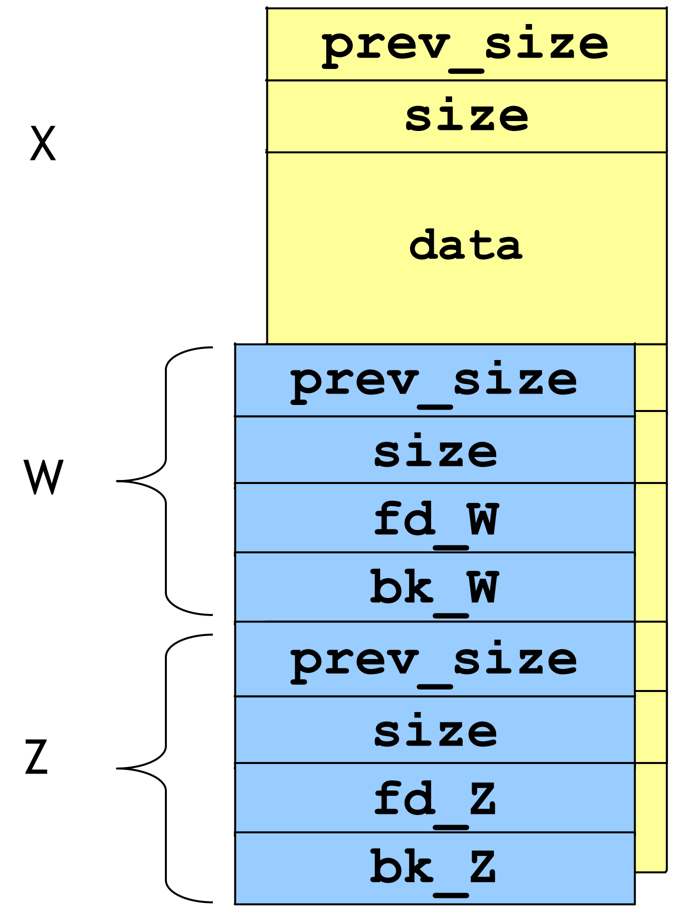
1. free(X) is called
2. W is examined and Z is found using (W + W->size)
   - Z says that W is free
3. unlink(W, fd_w, bk_w) is called

   *(fd_w + 12) = bk_w

   *(bk_w + 8) = fd_w

fd_w must be set to the address to be overwritten - 12

bk_w must be set to the value to be written (shellcode)

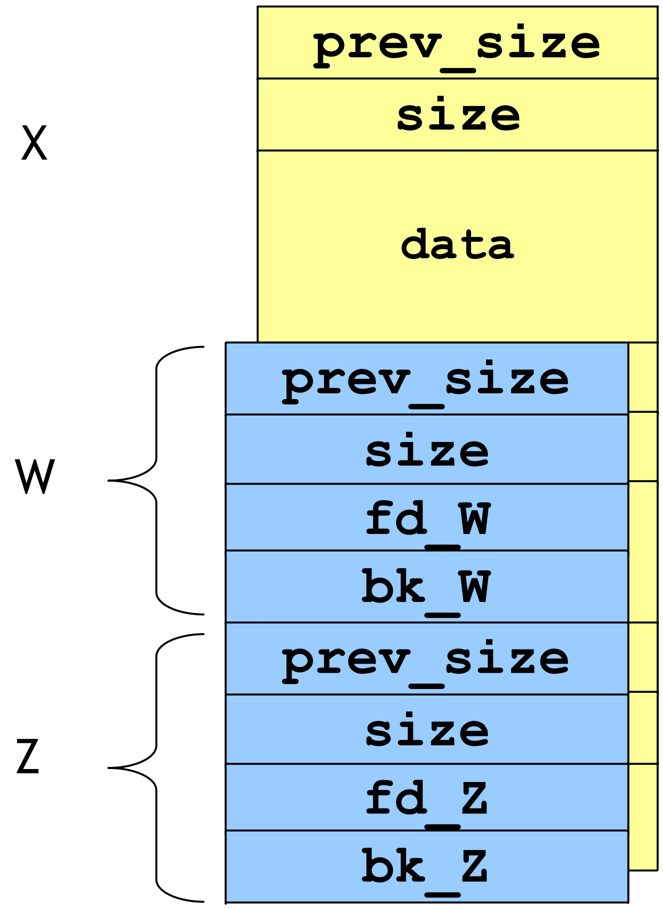


# Setting up the buffer

- Find address to overwrite (FUNC_POINTER)
  - E.g., address of GOT entry for free()
- First 8 bytes of X are overwritten by free
  - fd and bk pointers
  - We'll need to skip them
- Unlink modifies the BK + 8
  - We will also need to skip this

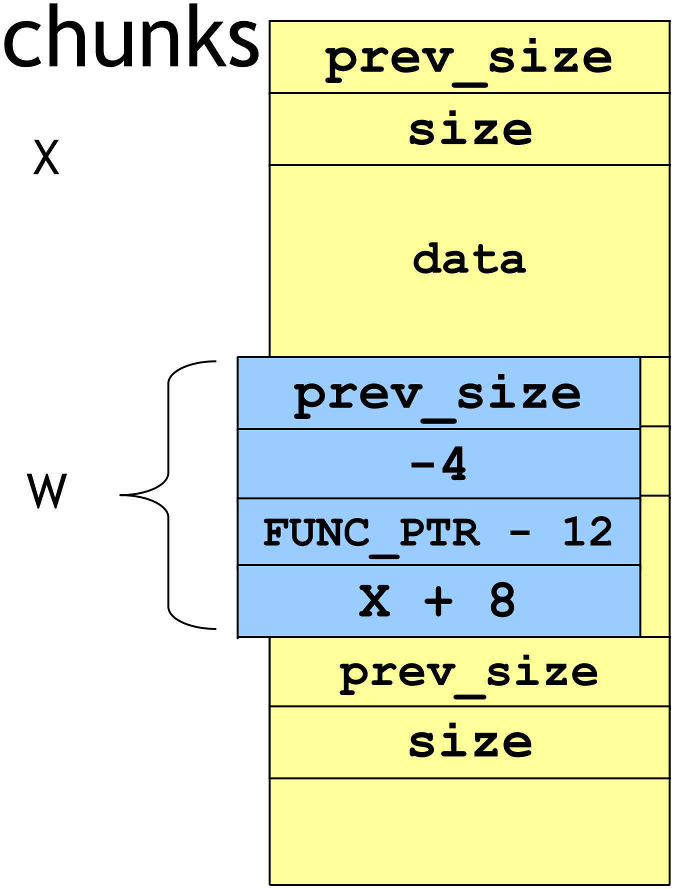
# Setting up the buffer

- Target address: 8 bytes into X
  - Skip new fd and bk set by free
- Jump instructions (0xeb 0xc) to jump over the part of the buffer modified by unlink
- Actual shellcode
- Padding to overwrite Y
- Fake chunk (W)
  - fd= FUNC_POINTER – 12
  - bk= X + 8
- Fake chunk (Z)
  - PREV_INUSE = 0



# Neat trick

- We don’t need two fake chunks
- Fake chunk W
  - size = -4 (`0xfffffffc`)



# Take away points

- Another example of problems arising from storing control information in-band
  - As when storing return address on the stack
- Exploiting certain vulnerabilities requires a detailed understanding of low-level implementation details