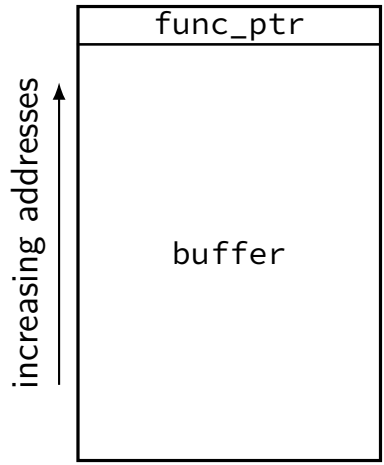
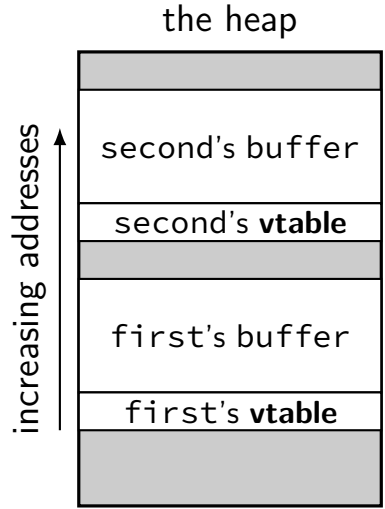
easy heap overflows

```
struct foo {
    char buffer[100];
    void (*func_ptr)(void);
};
```



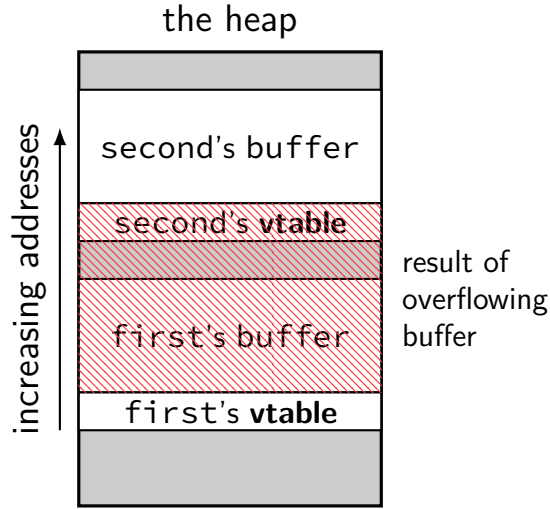
heap overflow: adjacent allocations

```
class V {
  char buffer[100];
public:
  virtual void ...;
  ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



heap overflow: adjacent allocations

```
class V {
  char buffer[100];
public:
  virtual void ...;
  ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



heap structure

where does malloc, free, new, delete, etc. keep info?

often in data structures next to objects on the heap

special case of adjacent heap objects problem

topic for later

sudo exploit

this writeup: summary from `https://www.openwall.com/lists/oss-security/2021/01/26/3`

from group at Qualys

sudo bug

the bug:

```
for (size = 0, av = NewArgv + 1; *av; av++)
     size += strlen(*av) + 1;
if (size == 0 || (user_args = malloc(size)) == NULL) { ... }
...
for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
while (*from) {
  if (from[0] == '\\' && !isspace((unsigned char)from[1]))
    from++;
  *to++ = *from++;
...
```

can skip `\0` if prefixed with backslash

but `strlen` used to allocate buffer

disagreement about copied string length

brute-forcing?

method: tried to lots of buffer overflows, get crashes

looked at them by hand, found interesting ones...

one crash

```
0x000056291a25d502 in process_hooks_getenv (name=name@...ry=0x7f4a6d7dc046 "SYSTEMD_BYPASS_U

=> 0x56291a25d502 <process_hooks_getenv+82>:    callq  *0x8(%rbx)

108         rc = hook->u.getenv_fn(name, &val, hook->closure);
```

they overwrote a function pointer on the heap!

next inquiry: where did that usually point?

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_po
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_po
```

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_po
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_po
```

observations (that hold true even with ASLR):

- addr(`execv@plt`) - addr(`sudoers_hook_getenv`) = `-0x6000`
- last 12 bits of execv@plt always `a00` (page alignment)

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

only four bits of random data from ASLR!

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

what if crashes? try again!

- would work about once every 16 tries...
- but actual exploit needed to write a 00 byte at the end (strcpy)
- so worked 'only' about once every 4096 tries

into exploit

make SYSTEMD_BYPASS_USERDB program in current directory

run sudo, triggering buffer overflow to change
`sudoers_hook_getenv("SYSTEMD_BYPASS_USERDB", ...)`
into
`execv(SYSTEMD_BYPASS_USERDB, ...)`

(well, try to change — it won't always work)

heap smashing

"lucky" adjancent objects

same things possible on stack

but stack overflows had nice generic "stack smashing"

is there an equivalent for the heap?

yes (mostly)

Linux memory allocation calls

`brk()`

- set 'break' at end of heap region
- one big memory region for dynamic memory
- used to be only way to allocate memory
- minimum size of changes = 4KB (x86-64)
- want larger changes for speed

`mmap()`

- allocate new memory region
- more complex OS bookkeeping than brk()
  - adding to list of memory regions, not changing a size
- minimum size = 4KB
- want much larger allocations for speed

malloc()/free()/etc. as memory partitioners

for “small” objects (less than kilobytes)

malloc() allocates *big chunks of memory*

then subdivides them on the fly

different strategies to do this

all need to track *metadata* about allocated objects!

malloc() metadata?

need to:

find unused chunks of memory

- even after A=malloc(), B=malloc(), C=malloc(), free(B)

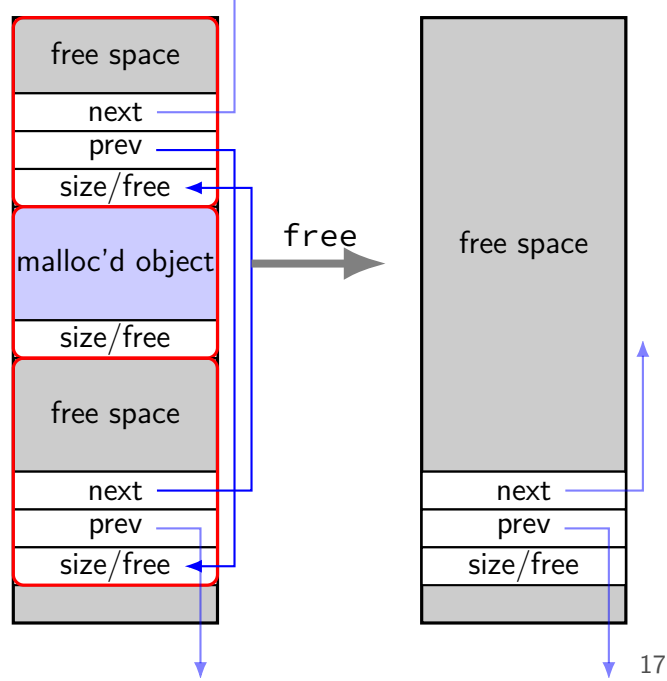
figure out how big allocation is when free() is called

two common strategies for tracking metadata:

- *before each allocation* (example: GNU libc malloc [Linux default])
- before 'arenas' of allocations (example: jemalloc [Firefox; *BSD default?])
  - lookup by rounding memory address

heap object

```
struct AllocInfo {
  bool free;
  int size;
  AllocInfo *prev;
  AllocInfo *next;
};
```



implementing free()

```
int free(void *object) {
  ...
  block_after = object + object_size;
  if (block_after->free) {
    /* unlink from list,
       prepare to merge with previous block */
    new_block->size += block_after->size;
    block_after->prev->next = block_after->next;
    block_after->next->prev = block_after->prev;
  }
  ...
}
```

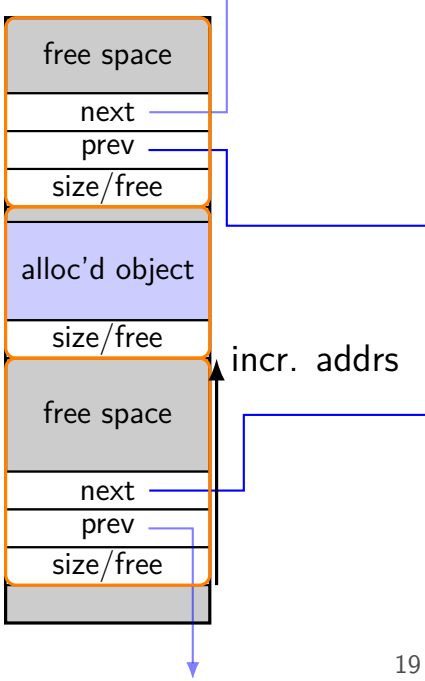
implementing free()

```
int free(void *object) {
  ...
  block_after = object + object_size;
  if (block_after->free) {
    /* unlink from list,
       prepare to merge with previous block */
    new_block->size += block_after->size;
    block_after->prev->next = block_after->next;
    block_after->next->prev = block_after->prev;
  }
  ...
}
```

arbitrary memory write

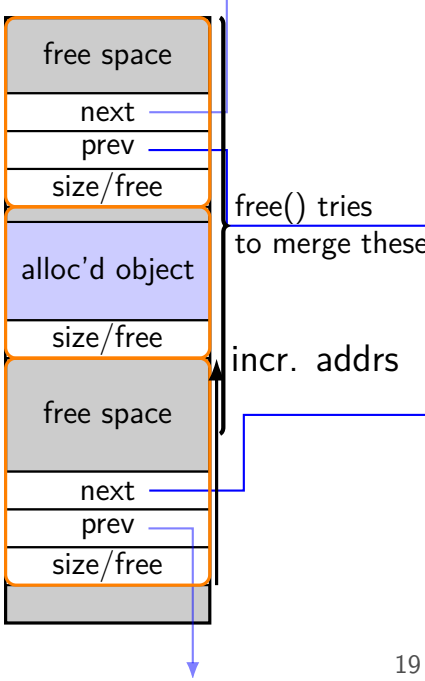
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



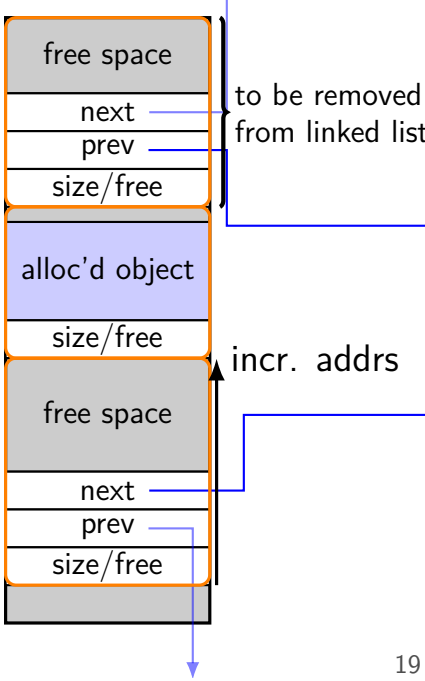
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



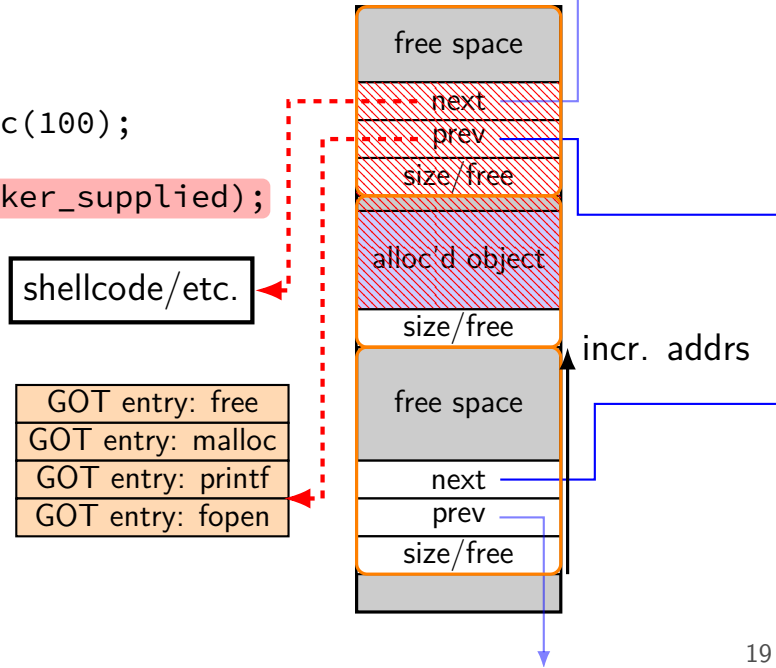
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



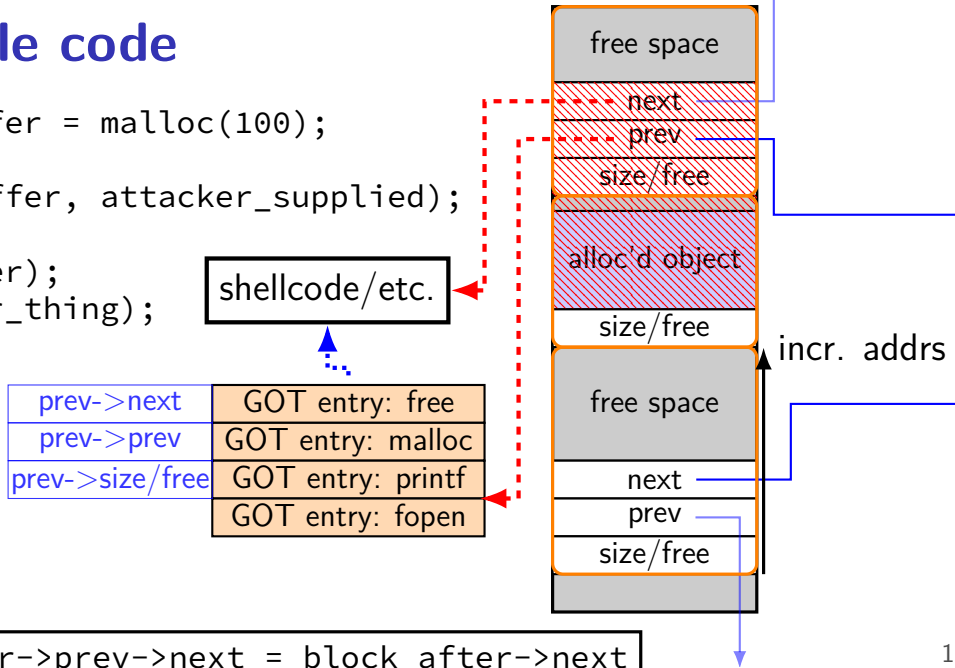
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



vulnerable code

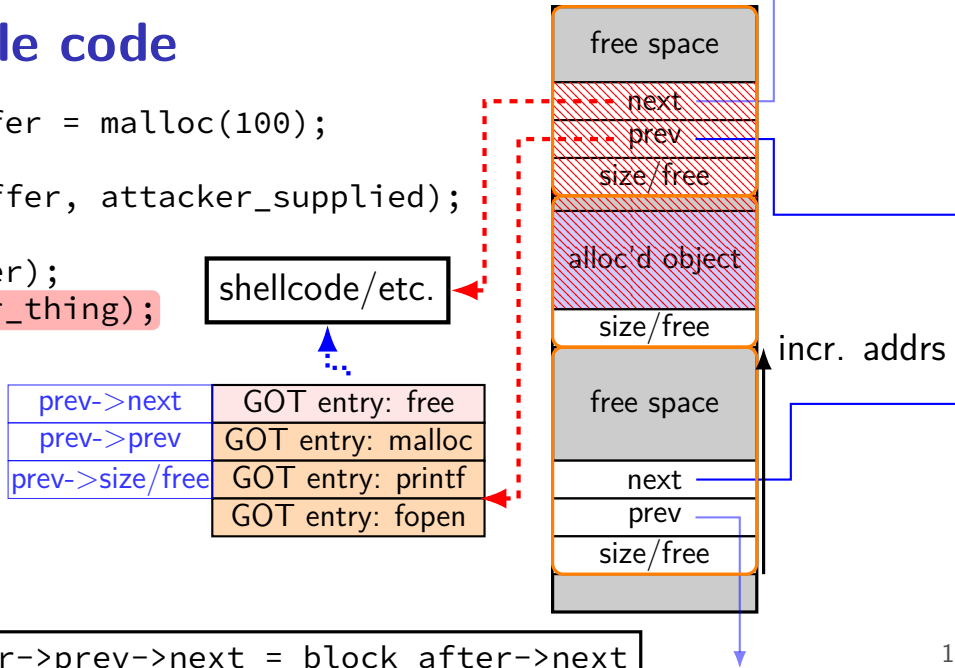
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```




```
block_after->prev->next = block_after->next
```

vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```

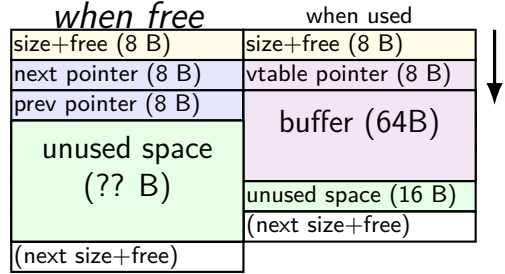



```
block_after->prev->next = block_after->next
```

heap overflow exercise

```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```

heap object layout

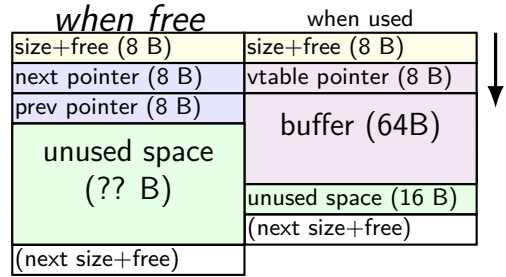


exercise 1:
to attack this buffer overflow
by overwriting the heap data structures
does it matter if space after a
is already free or not?

heap overflow exercise

```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```

heap object layout



exercise 2: if a at address 0x10000,
and attacker wants to overwrite
value at address 0x20000 with 0x30000,
where should attacker put 0x20000, 0x30000
in `attacker_controlled`?

other malloc designs?

there are a lot of different malloc/new implementations

often multiple free lists

free block list might not be kept with linked list

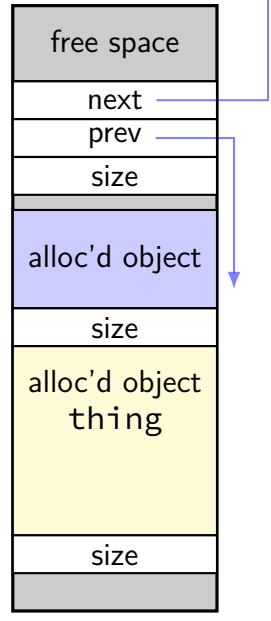
some place metadata next to allocations like this

some keep it separate

usually performance determines which is chosen

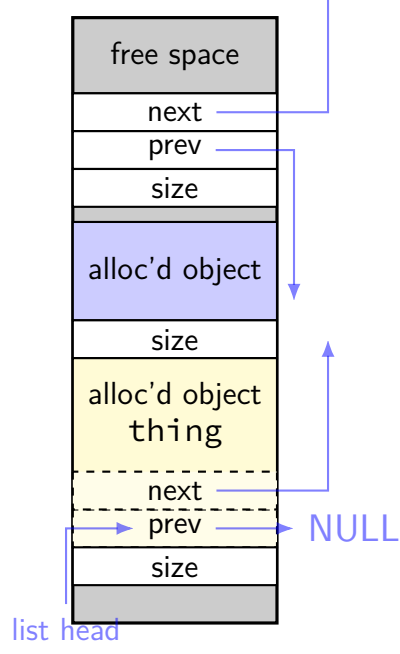
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



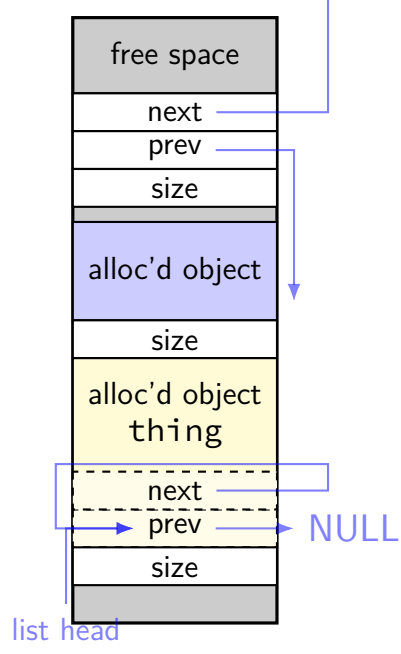
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```

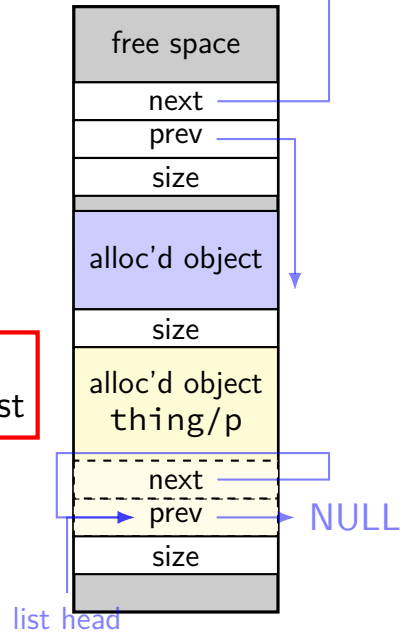


double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
```

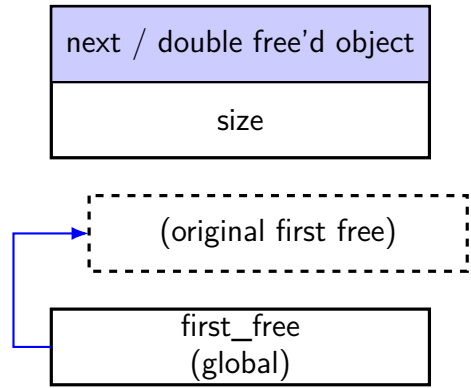
malloc returns something *still on free list*
because double-free made *loop* in linked list

```
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



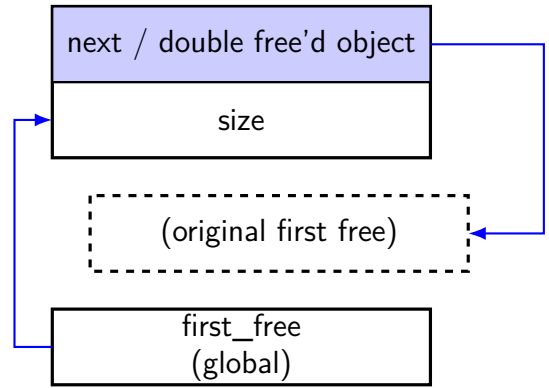
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



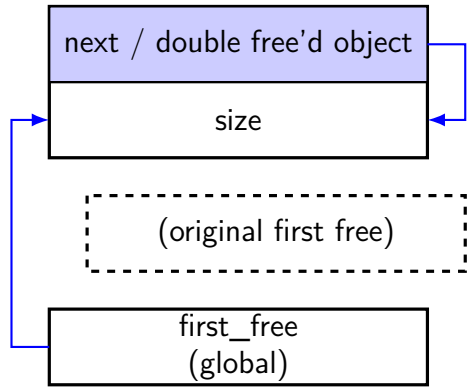
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



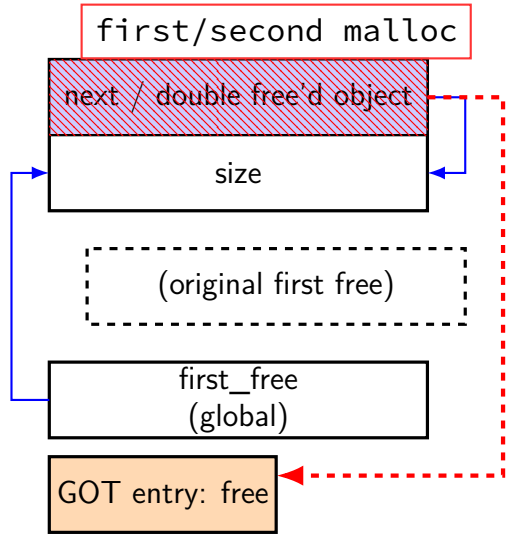
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



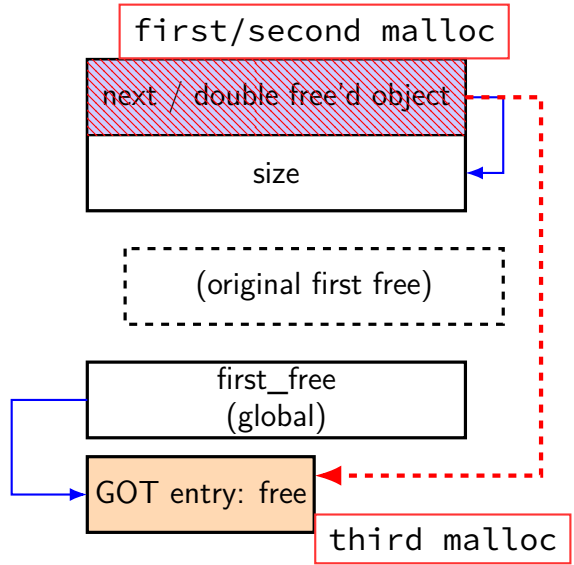
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free notes

this attack has apparently not been possible for a while

most malloc/new's *check for double-frees* explicitly

(e.g., look for a bit in `size` data)

prevents this issue — also catches programmer errors

pretty cheap

double-free exercise

```
free(...) {
    freed->next = first_free
    first_free = freed;
}
malloc(...) {
    if (can use first free) {
        void *to_return = first_free;
        first_free = first_free->next;
        return to_return;
    }
}
vulnerable() {
    char *p = malloc(100);
    free(p);
    free(p);
    char *q = malloc(100);
    char *r = malloc(100);
    strlcpy(q, attacker_input1, 100);
    char *s = malloc(100);
    strlcpy(r, attacker_input2, 100);
    strlcpy(s, attacker_input3, 100);
}
```

To do memory[0x123456] $\leftarrow$ 0x789abc
what should input1/input2/input3 be?

backup slides