Section 3: Advanced Buffer Overflow

CSE484

Including content from previous quarters by: Eric Zeng, Keanu Vestil, James Wang, Amanda Lam, Ivan Evtimov, Jared Moore, Franzi Roesner, Viktor Farkas

Administrivia

Lab 1a due Jan 22th @ 11:59pm

Final deadline for Lab 1b is January 29th @ 11:59pm

Lab 1 Notes/Hints

- If you get stuck, move on!
- Don’t procrastinate on Sploits 5-7 (some of them are harder)

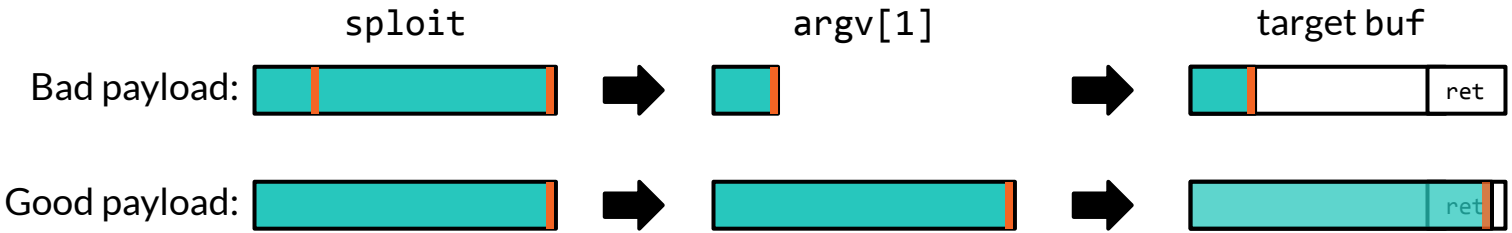
Sploit 3

- No frame pointer (EBP), so you can only change last byte of saved return address (EIP).
- Hint - In a stack frame, your shellcode can appear in two places:
  1) A pointer to the shellcode in the arguments section of the stack frame
  2) In the buffer that the target program copies the shellcode to

A Note About Null

Your payload is treated as a string.

- Null byte (`\x00`) can terminate it early
- Changing buffer size will shift addresses
- Double check memory



strcpy: I’m going to keep copying bytes until I see NULL

you:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89
\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8
\xdc\xff\xff\xff/bin/sh\x90\x90\x90\x90\x90...
```

strcpy:

Why do we care about buffer overflows?

- Notable malware that used buffer overflow exploits
  - SQL Slammer worm (2003)
    - Buffer overflow vulnerability in MS SQL Server, attacked open UDP ports
    - Infected 75000 computers in 10 minutes, took down numerous routers
  - WannaCry and NotPetya ransomware (2017)
    - Uses exploit in MS Windows sharing protocol, called *EternalBlue*, developed by NSA
    - Used to enable malware that encrypts a computer's files and ransom them for BTC
    - Affected many people, large companies, caused $billions in damages
- Most security bugs in large C/C++ codebases are due to memory corruption vulns
  - Google: "Our data shows that issues like use-after-free, double-free, and heap buffer overflows generally constitute more than 65% of High & Critical security bugs in Chrome and Android."
  - Microsoft: "~70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues"
  - Read more: https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/

memory unsafe languages (C, C++)

Rust, Go

Further reading:
https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering

Useful resources/tools:

- Aleph One “Smashing the Stack for Fun and Profit” (also see: “revived version”)
- scut “Exploiting Format String Vulnerabilities”
- Chien & Ször “Blended attack exploits…”
- Office Hours
- Ed Discussion Board

Sploit 5??

➔ **What makes it different?**

Buffer copied to the heap (instead of stack)

➔ **What makes it vulnerable?**

The behavior of freeing an already freed memory chunk is undefined [Commonly known as double-free]

➔ **Useful Resources**

Read "Once upon a free()"

[http://phrack.org/issues/57/9.html]

Dynamic Memory Management in C

- Memory allocation: **`malloc(size_t n)`**
  - Allocates n bytes (doesn't clear memory)
  - Returns a pointer to the allocated memory

- Memory deallocation: **`free(void* p)`**
  - Frees the memory space pointed to by p
  - p must have been returned by a previous call to `malloc()` (or similar).
  - If p is null, no operation is performed.
  - If `free(p)` has been called before ("double free"), undefined behavior occurs.

`tmalloc` implementation

- We provide an implementation of malloc in `tmalloc.c` and use that in `target5`.
- Note that `tmalloc.c` does not use the actual heap!
- Common in embedded devices with an OS that doesn't have a heap.
- We allocate our own space in the global variables region that we manage with `tmalloc`, `tfree`, `trealloc`, etc. as if though it's a heap.
- Line 57: `static CHUNK arena[ARENA_CHUNKS];`

Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation.

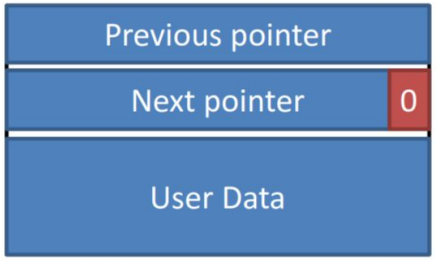
`tmalloc` and Chunks

Note: the free bit is stored in the same 4 byte word as the next pointer.

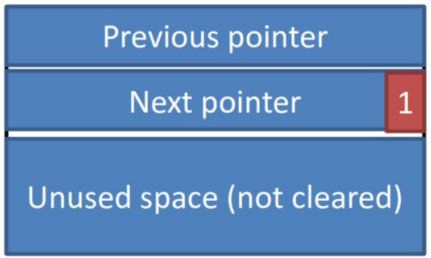
This is possible because tmalloc chunks are aligned on 8 byte word boundaries, so we know that the last bit is never used to refer to an address.

In binary:
0x0: 00000
0x8: 01000

- Chunks of heap memory are organized into a doubly-linked list
- Each chunk contains pointers to the next and previous chunk in the list.
- The least significant bit of the next pointer is the “free bit”



Allocated Chunk

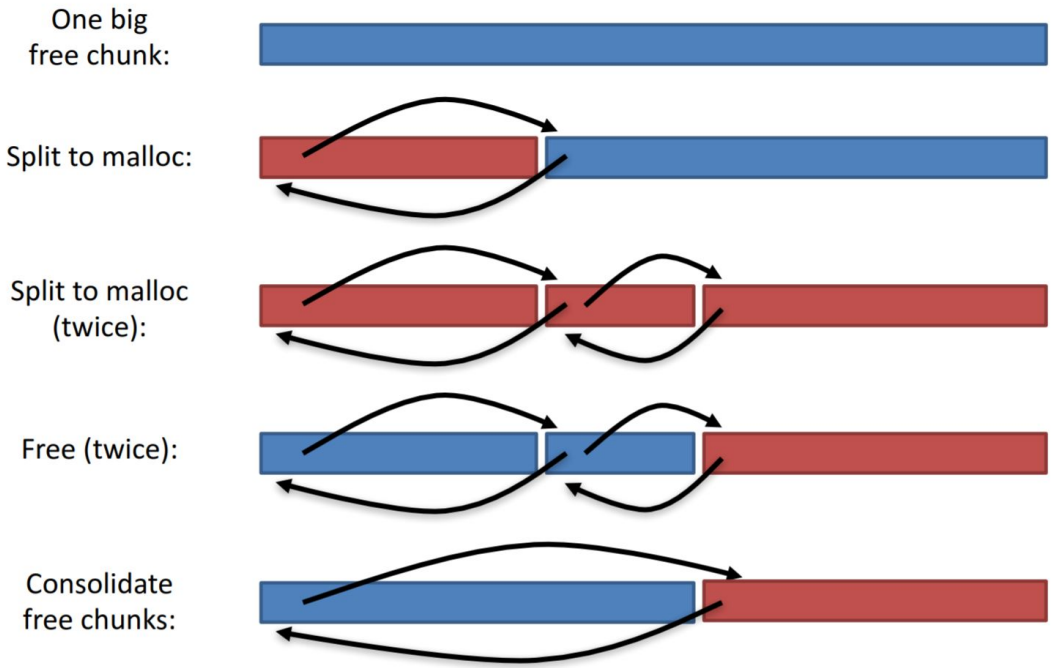


Free Chunk

Chunk header definition


```
15  /*
16   * the chunk header
17   */
18  typedef double ALIGN;
19
20  typedef union CHUNK_TAG
21  {
22    struct
23      {
24        union CHUNK_TAG *l;       /* leftward chunk */
25        union CHUNK_TAG *r;       /* rightward chunk + free bit (see below) */
26      } s;
27    ALIGN x;
28  } CHUNK;
29
30  /*
31   * we store the freebit -- 1 if the chunk is free, 0 if it is busy --
32   * in the low-order bit of the chunk's r pointer.
33   */
34
```

Chunk Maintenance



Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation.

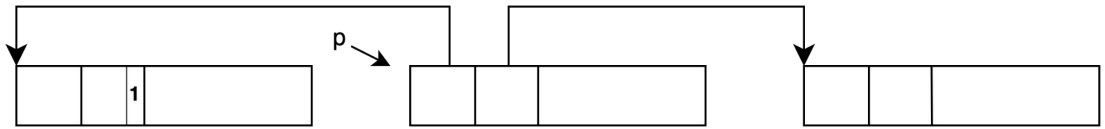
Activity: Understanding tfree

Given the following code from tfree, draw what happens when tfree(p) is called.

```
#define SET_FREEBIT(chunk) ( *(unsigned *)&(chunk)->s.r |=  0x1 )
#define CLR_FREEBIT(chunk) ( *(unsigned *)&(chunk)->s.r &= ~0x1 )
#define GET_FREEBIT(chunk) ( (unsigned)(chunk)->s.r & 0x1 )
```

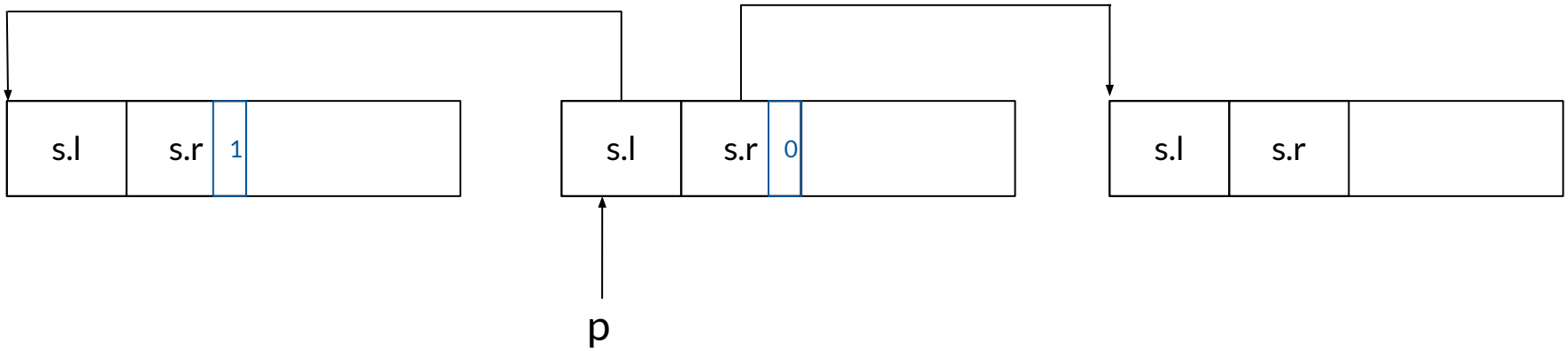
(The first chunk of memory is free)

```
108    q = p->s.l;
109    if (q != NULL && GET_FREEBIT(q))
110      {
111        CLR_FREEBIT(q);
112        q->s.r      = p->s.r;
113        p->s.r->s.l = q;
114        SET_FREEBIT(q);
115        p = q;
116      }
```



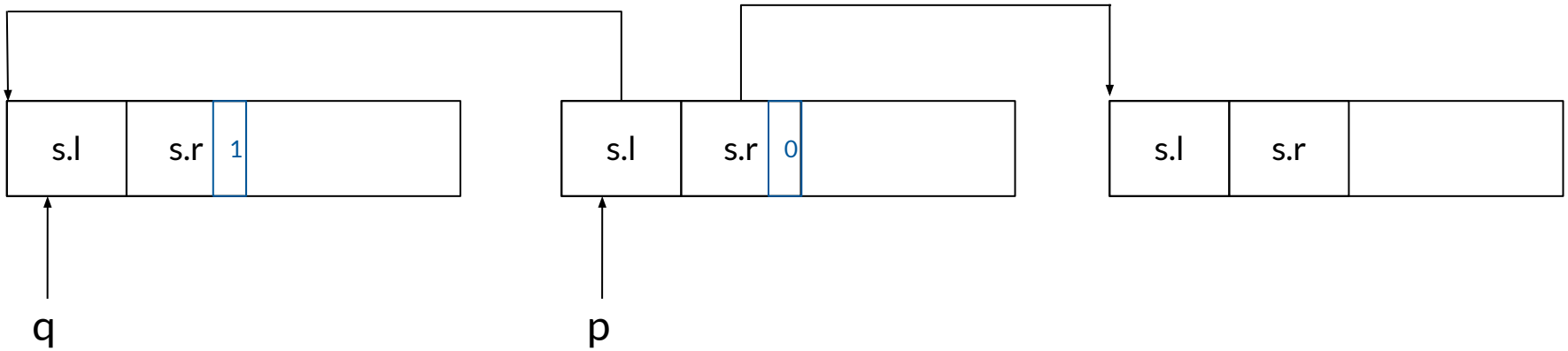
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



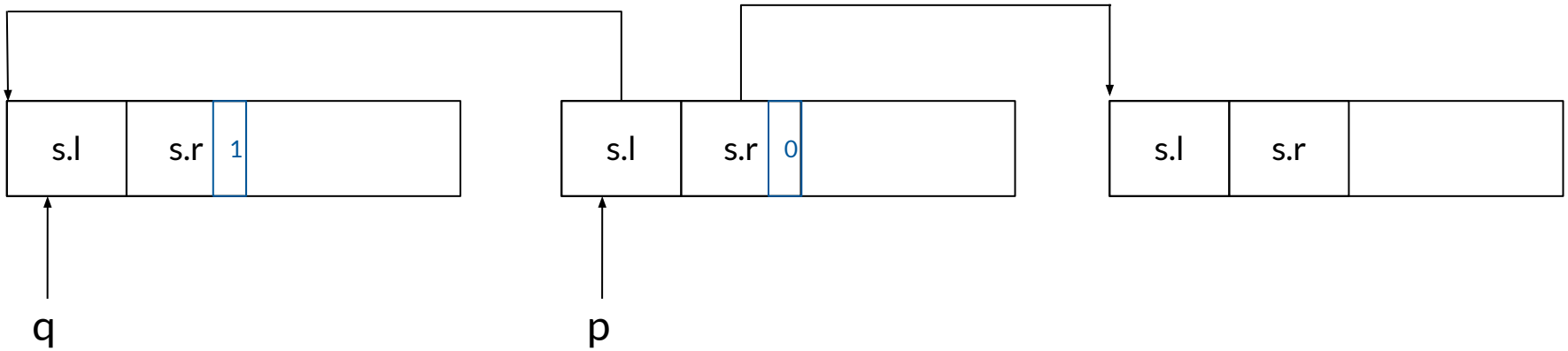
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



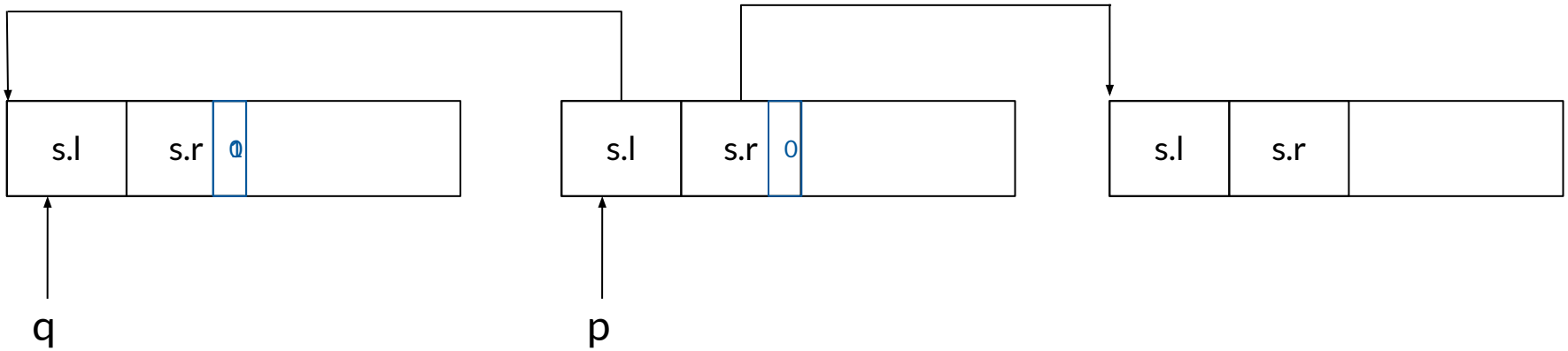
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



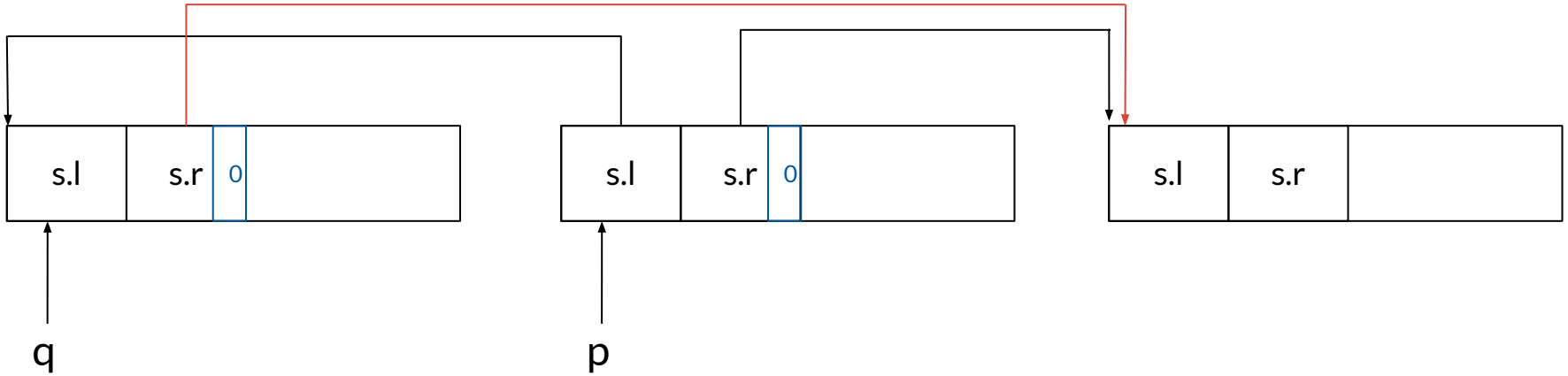
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



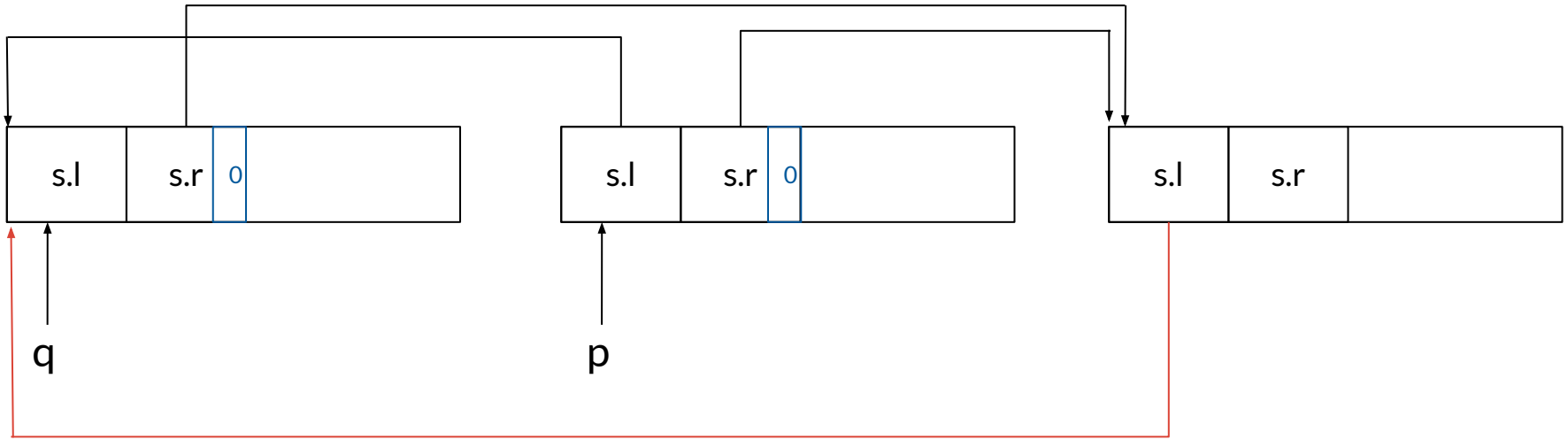
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



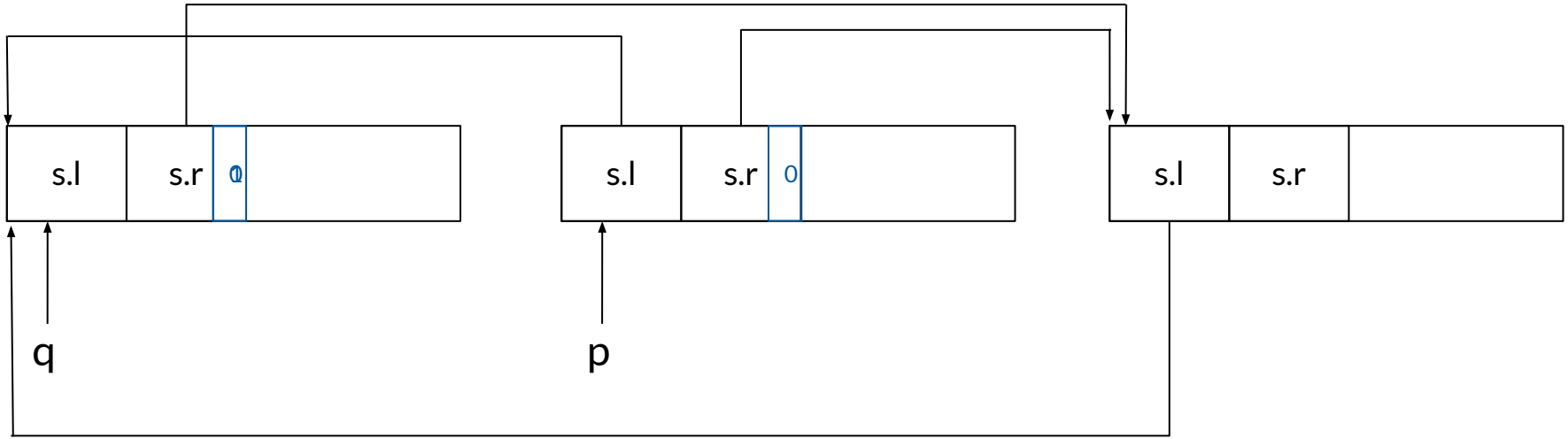
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



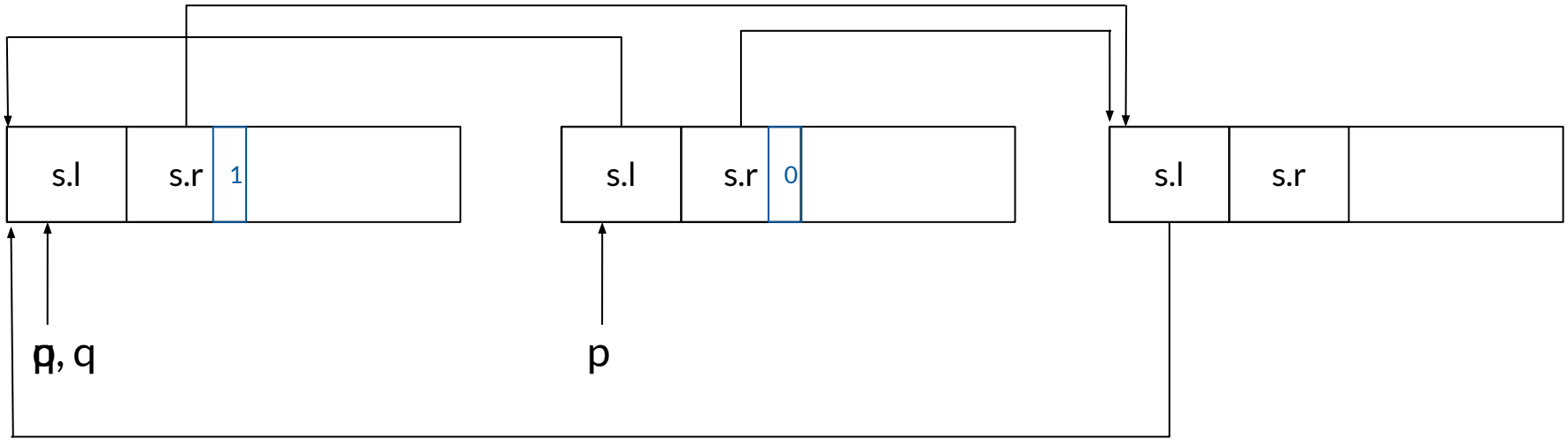
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



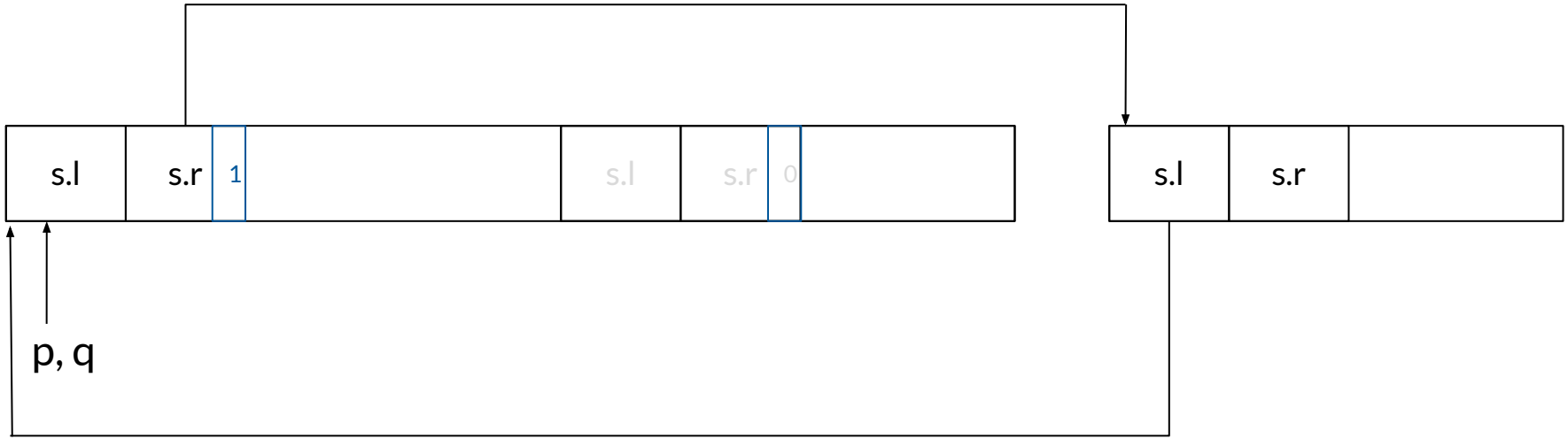
```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free



```
108   q = p->s.l;
109   if (q != NULL && GET_FREEBIT(q))
110     {
111       CLR_FREEBIT(q);
112       q->s.r      = p->s.r;
113       p->s.r->s.l = q;
114       SET_FREEBIT(q);
115       p = q;
116     }
```

0 = in use
1 = free

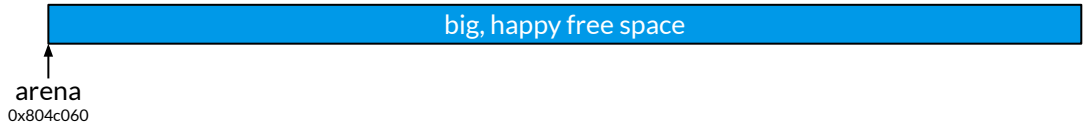


tmalloc.h usage example

```
1.  int main(){
2.      char* dyn;
3.      char* input =
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa
    8\xa9";

1.      dyn = tmalloc(10);

1.      if(dyn == NULL){
2.          fprintf(stderr,
    “err\n");
3.          exit(EXIT_FAILURE);
4.      }

1.      memcpy(dyn, input, 10);

1.      tfree(dyn);

1.      return  0;
2.  }
```

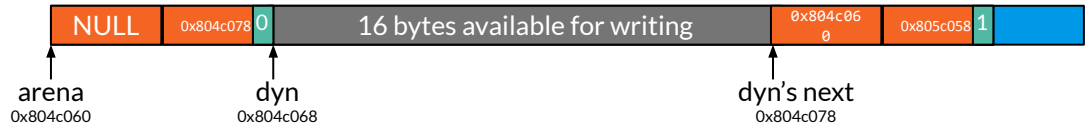
Before `tmalloc` call (line 4):

```
0x804c060 <arena>:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070 <arena+16>:   0x00000000      0x00000000      0x00000000      0x00000000
0x804c080 <arena+32>:   0x00000000      0x00000000      0x00000000      0x00000000
0x804c090 <arena+48>:   0x00000000      0x00000000      0x00000000      0x00000000
```



After `tmalloc` call: chunk pointers created

```
0x804c060 <arena>:      0x00000000      0x0804c078      0x00000000      0x00000000
0x804c070 <arena+16>:   0x00000000      0x00000000      0x0804c060      0x0805c059
0x804c080 <arena+32>:   0x00000000      0x00000000      0x00000000      0x00000000
0x804c090 <arena+48>:   0x00000000      0x00000000      0x00000000      0x00000000
```



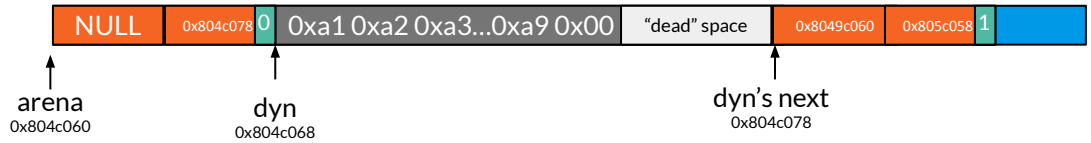
Printed with: x /16xw arena

tmalloc.h usage example

```
1.  int main(){
2.      char* dyn;
3.      char* input =
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa
    8\xa9";

1.      dyn = tmalloc(10);

1.      if(dyn == NULL){
2.          fprintf(stderr,
    “err\n");
3.          exit(EXIT_FAILURE);
4.      }

1.      memcpy(dyn, input, 10);

1.      tfree(dyn);

1.      return  0;
2.  }
```

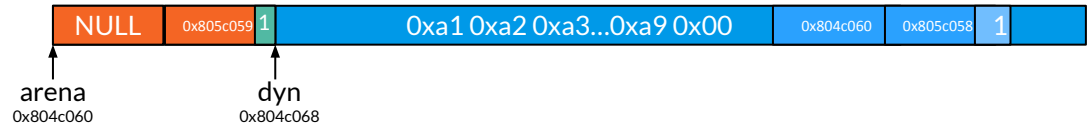
After the copy in line 9:

```
0x804c060 <arena>:     0x00000000     0x0804c078     0xa4a3a2a1     0xa9a8a7a5
0x804c070 <arena+16>:  0x000000a9     0x00000000     0x0804c060     0x0805c059
0x804c080 <arena+32>:  0x00000000     0x00000000     0x00000000     0x00000000
0x804c090 <arena+48>:  0x00000000     0x00000000     0x00000000     0x00000000
```



After the tfree, the chunk is coalesced (line 10)

```
0x804c060 <arena>:     0x00000000     0x0805c059     0xa4a3a2a1     0xa9a8a7a5
0x804c070 <arena+16>:  0x000000a9     0x00000000     0x0804c060     0x0805c059
0x804c080 <arena+32>:  0x00000000     0x00000000     0x00000000     0x00000000
0x804c090 <arena+48>:  0x00000000     0x00000000     0x00000000     0x00000000
```



Printed with: x /16xw arena

```
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);

  tfree(b);

  return 0;
}
```

Target 5

- BUFLEN = 120
- Copies your buffer into heap memory allocated by tmalloc()
- What's the vulnerability?

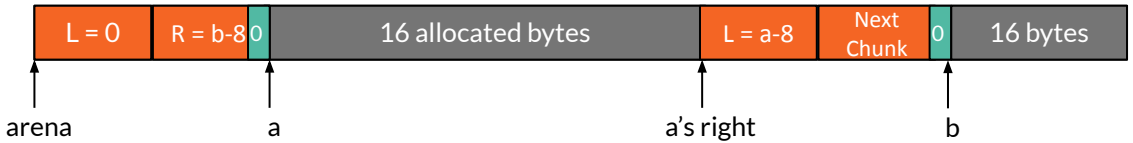
b is freed twice, but only allocated once

Double `tfree` example

```
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
  {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);    }

  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);

  tfree(b);

  return 0;
}
```

After `tmalloc` call for b:

```
0x8049da0 <arena>:     0x00000000  0x08049db8  0x00000000  0x00000000
0x8049db0 <arena+16>:  0x00000000  0x00000000  0x08049da0  0x08049dd0
0x8049dc0 <arena+32>:  0x00000000  0x00000000  0x00000000  0x00000000
0x8049dd0 <arena+48>:  0x08049db8  0x08059d99
```



After `tfree` call for a:

```
0x8049da0 <arena>:     0x00000000  0x08049db9  0x00000000  0x00000000
0x8049db0 <arena+16>:  0x00000000  0x00000000  0x08049da0  0x08049dd0
0x8049dc0 <arena+32>:  0x00000000  0x00000000  0x00000000  0x00000000
0x8049dd0 <arena+48>:  0x08049db8  0x08059d99
```

Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

Double `tfree` example

```
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
  {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);

  tfree(b);

  return 0;
}
```

After `tfree` call for a:

```
0x8049da0 <arena>:     0x00000000  0x08049db9  0x00000000  0x00000000
0x8049db0 <arena+16>:  0x00000000  0x00000000  0x08049da0  0x08049dd0
0x8049dc0 <arena+32>:  0x00000000  0x00000000  0x00000000  0x00000000
0x8049dd0 <arena+48>:  0x08049db8  0x08059d99
```

After `tfree` call for b:

```
0x8049da0 <arena>:     0x00000000  0x08049dd1  0x00000000  0x00000000
0x8049db0 <arena+16>:  0x00000000  0x00000000  0x08049da0  0x08049dd0
0x8049dc0 <arena+32>:  0x00000000  0x00000000  0x00000000  0x00000000
0x8049dd0 <arena+48>:  0x08049db8  0x08059d99
```

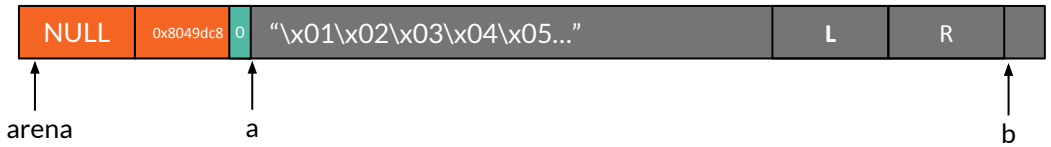
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

Double `tfree` example

```
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
  {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);

  tfree(b);

  return 0;
}
```

Our input buffer contains: \x01\x02\x03…\x0f\x10\x11\x12\x13
After copying the buffer to the new a:

```
0x8049da0 <arena>:      0x00000000      0x08049dc8      0x04030201      0x08070605
0x8049db0 <arena+16>:   0x0c0b0a09      0x100f0e0d      0x00131211      0x08049dd0
0x8049dc0 <arena+32>:   0x00000000      0x00000000      0x08049da0      0x08059d99
0x8049dd0 <arena+48>:   0x08049db8      0x08059d99
```



What are the contents of L, the word that used to be a pointer to b's left?

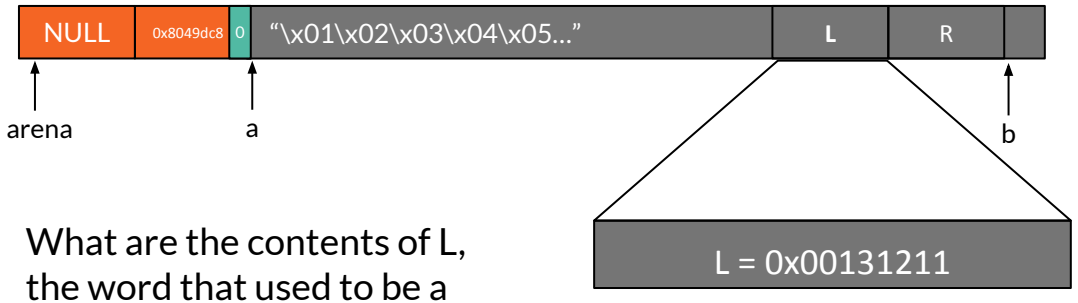
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

Double `tfree` example

```
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
  {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);

  tfree(b);

  return 0;
}
```

Our input buffer contains: \x01\x02\x03...\x0f\x10\x11\x12\x13
After copying the buffer to the new a:

```
0x8049da0 <arena>:      0x00000000  0x08049dc8  0x04030201  0x08070605
0x8049db0 <arena+16>:   0x0c0b0a09  0x100f0e0d  0x00131211  0x08049dd0
0x8049dc0 <arena+32>:   0x00000000  0x00000000  0x08049da0  0x08059d99
0x8049dd0 <arena+48>:   0x08049db8  0x08059d99
```



What are the contents of L, the word that used to be a pointer to b's left?

Exploit hint 1: We can control the value stored at `b->s.l`!

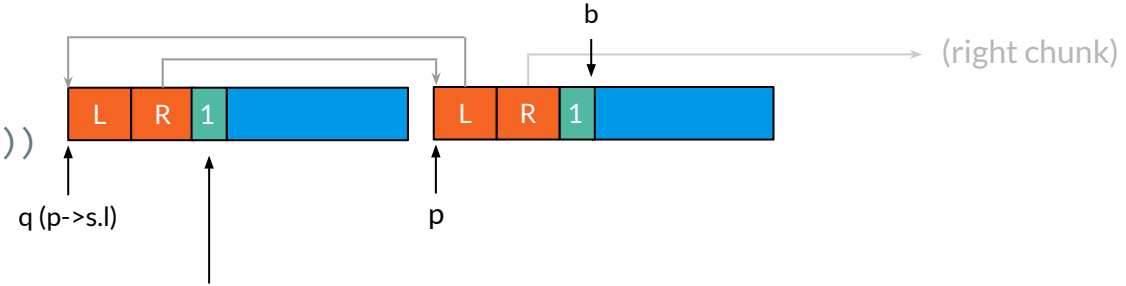
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

Double `tfree` example

What would happen in `tfree(b)`?

```
    [...]
106  p = TOCHUNK(vp);
107  CLR_FREEBIT(p);
108  q = p->s.l;
/* try to consolidate leftward */
109  if (q != NULL && GET_FREEBIT(q))
110    {
111      CLR_FREEBIT(q);
112      q->s.r      = p->s.r;
113      p->s.r->s.l = q;
114      SET_FREEBIT(q);
115      p = q;
116    }
    [...]
```

At line 108, `tfree` assigns the variable `q` to `p`'s left chunk (`p->s.l`). Then, it checks if the chunk at `q` is free, and merges the chunks if it is free



To trigger the chunk merge, we need to be sure q's free bit is set to (1).

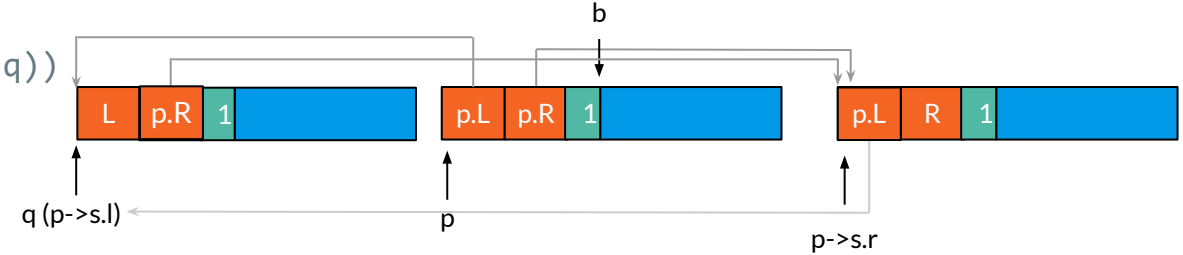
Double `tfree` example

What would happen in `tfree(b)`?

```
     [...]
106  p = TOCHUNK(vp);
107  CLR_FREEBIT(p);
108  q = p->s.l;
/* try to consolidate leftward */
109  if (q != NULL && GET_FREEBIT(q))
110    {
111      CLR_FREEBIT(q);
112      q->s.r      = p->s.r;
113      p->s.r->s.l = q;
114      SET_FREEBIT(q);
115      p = q;
116    }
     [...]
```

Note: While the slides use p.R and p.L, this is shorthand for referring to p's right and left pointers, and is NOT C code. (To reference p's right and left pointers, we will need to first dereference the struct s, then access the left pointer).

Line 112: `tfree` sets `q.r` to the address of `p`'s right chunk
Line 113: `tfree` copies the address of `q` to `p`'s right chunk's left/prev pointer (`p->s.r->s.l`)



What if `p.r` and `p.l` didn't point to real chunks?

Exploit hint 2: Can overwrite a location (`p.r.l`) with a value we specified (which `tfree` sets by reading `p.l`)

What if p.r = &RET, and p.l = &buf?

Final Words

- Good luck on lab 1b, please start early!!

- Post questions on discussion board