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 Tomorrow, April 14th, @ 11:59pm
  - Can use late days, max 3 per assignment (5 late days total)
  - Use the `turnin.sh` script to save sploits 1-3
  - You are not allowed to modify the content of exploits after running the script (feel free to save copies of your sploits 1-3 just in case)
- Deadline for Lab1b is April 24th @ 11:59pm

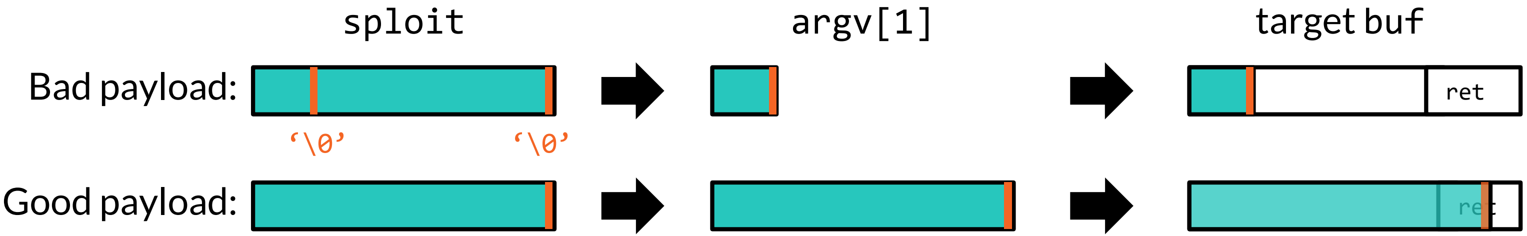
Lab 1 Notes/Hints

- If you get stuck, move on!
- Don't procrastinate on Sploits 4-7. (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) 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 shellcode 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++, assembly)

memory safe languages

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.

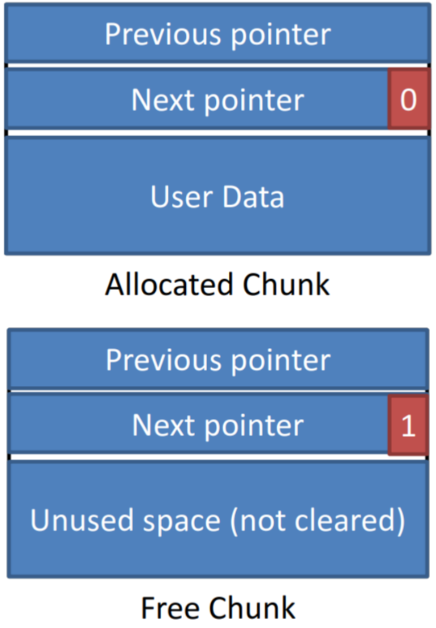
`tmalloc` and Chunks

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: 0b0000
0x8: 0b1000

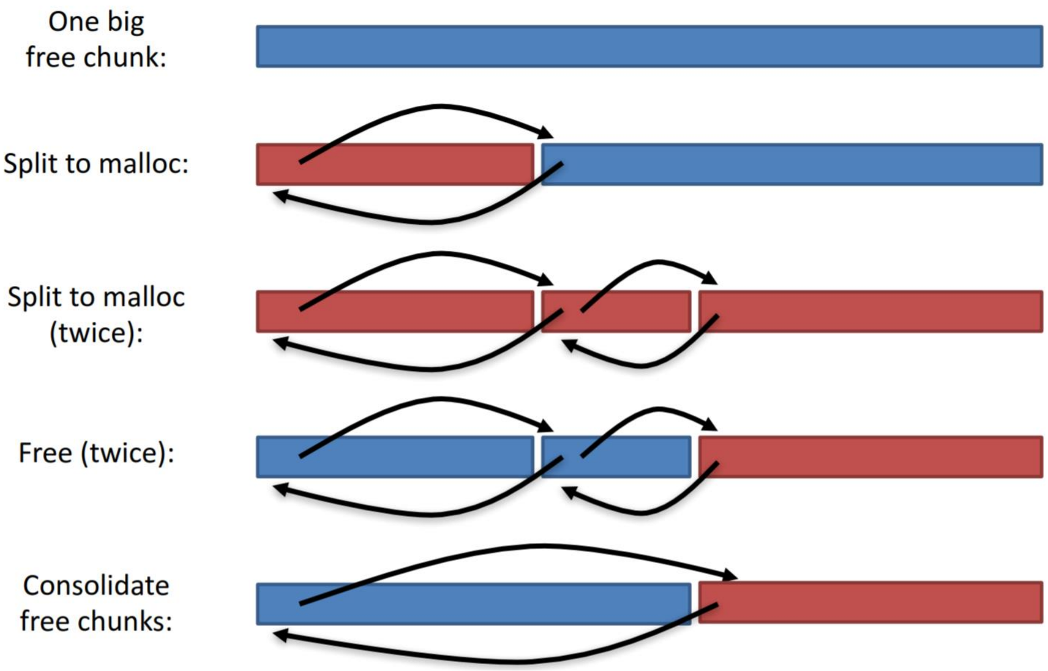
- Chunks of heap memory are organized into a doubly-linked list
- Each chunk contains pointers to the next and previous chunk in the list.
- The least significant byte of the next pointer contains the "free bit"



Chunk header definition


```c
/*
 * the chunk header
 */
typedef double ALIGN;

typedef union CHUNK_TAG
{
  struct
    {
      union CHUNK_TAG *l;       /* leftward chunk */
      union CHUNK_TAG *r;       /* rightward chunk + free bit (see below) */
    } s;
  ALIGN x;
} CHUNK;

/*
 * we store the freebit -- 1 if the chunk is free, 0 if it is busy --
 * in the low-order bit of the chunk's r pointer.
 */

```

Chunk Maintenance



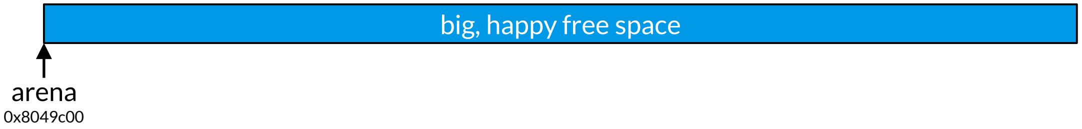
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation.

`tmalloc.h` usage example

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <sys/types.h>
 5  #include "tmalloc.h"
 6
 7  int main(int argc, char *argv[]){
 8      //we will hold the heap-allocated pointer here
 9      char* p;
10
11      //we will copy this into the heap memory
12      //currently, it's stupid to have it both on the stack
13      //and on the heap, but this is just a demonstration
14->     char* buf = "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9";
15
16      if ( (p = tmalloc(10)) == NULL)
17      {
18        fprintf(stderr, "tmalloc failure\n");
19        exit(EXIT_FAILURE);
20      }
21
22      memcpy(p, buf, 10);
23
24      tfree(p);
25
26     return 0;
27  }
```

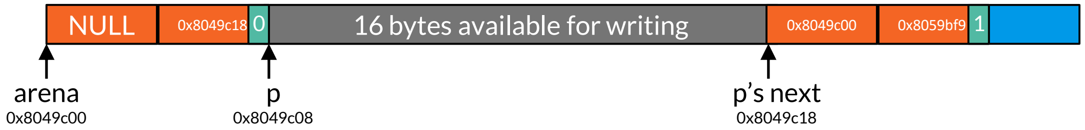
Before `tmalloc` call (line 16):

```
(gdb) x /15xw arena
0x8049c00 <arena>:      0x00000000   0x00000000   0x00000000   0x00000000
0x8049c10 <arena+16>:   0x00000000   0x00000000   0x00000000   0x00000000
0x8049c20 <arena+32>:   0x00000000   0x00000000   0x00000000   0x00000000
0x8049c30 <arena+48>:   0x00000000   0x00000000   0x00000000
```



After `tmalloc` call: chunk pointers created

```
(gdb) x /15xw arena
0x8049c00 <arena>:      0x00000000   0x08049c18   0x00000000   0x00000000
0x8049c10 <arena+16>:   0x00000000   0x00000000   0x08049c00   0x08059bf9
0x8049c20 <arena+32>:   0x00000000   0x00000000   0x00000000   0x00000000
0x8049c30 <arena+48>:   0x00000000   0x00000000   0x00000000
(gdb)
```



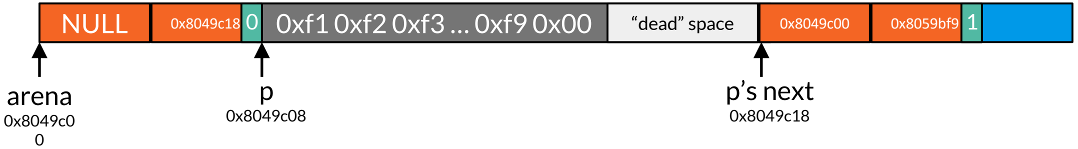
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation.

`tmalloc.h` usage example

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include "tmalloc.h"
6
7  int main(int argc, char *argv[]){
8      //we will hold the heap-allocated pointer here
9      char* p;
10
11     //we will copy this into the heap memory
12     //currently, it's stupid to have it both on the stack
13     //and on the heap, but this is just a demonstration
14 ->  char* buf = "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9";
15
16     if ( (p = tmalloc(10)) == NULL)
17     {
18       fprintf(stderr, "tmalloc failure\n");
19       exit(EXIT_FAILURE);
20     }
21
22     memcpy(p, buf, 10);
23
24     tfree(p);
25
26    return 0;
27 }
```

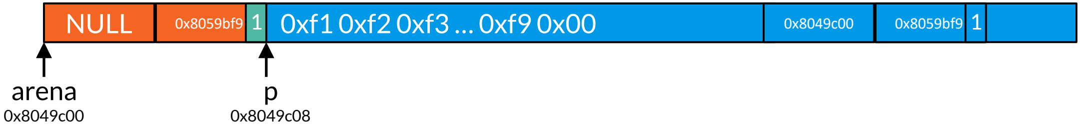
After the user writes in line 22 (note little-endianness in printout):

```
(gdb) x /15xw arena
0x8049c00 <arena>:      0x00000000      0x08049c18      0xf4f3f2f1      0xf8f7f6f5
0x8049c10 <arena+16>:   0x000000f9      0x00000000      0x08049c00      0x08059bf9
0x8049c20 <arena+32>:   0x00000000      0x00000000      0x00000000      0x00000000
0x8049c30 <arena+48>:   0x00000000      0x00000000      0x00000000
```



When `tfree` is called, this chunk is coalesced with the next one :

```
(gdb) x /15xw arena
0x8049c00 <arena>:      0x00000000      0x08059bf9      0xf4f3f2f1      0xf8f7f6f5
0x8049c10 <arena+16>:   0x000000f9      0x00000000      0x08049c00      0x08059bf9
0x8049c20 <arena+32>:   0x00000000      0x00000000      0x00000000      0x00000000
0x8049c30 <arena+48>:   0x00000000      0x00000000      0x00000000
(gdb)
```



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

```c
int foo(char *arg)
{
  char *p;
  char *q;

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

  tfree(p);
  tfree(q);

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

  obsd_strlcpy(p, arg, BUFLEN * 2);

  tfree(q);

  return 0;
}
```

Target 5

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

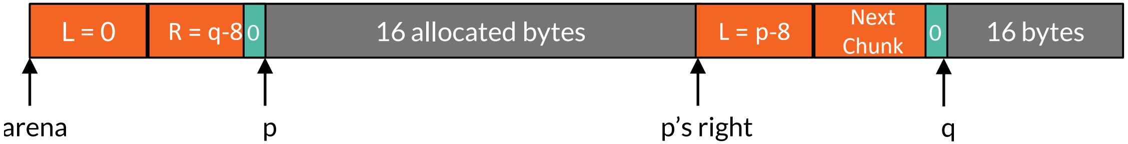
q is freed twice, but only allocated once

```c
46 int foo(char *arg)
47 {
48   char *p;
49   char *q;
50
51   if ( (p = tmalloc(16)) == NULL)
52     {
53       fprintf(stderr, "tmalloc failure\n");
54       exit(EXIT_FAILURE);
55     }
56   if ( (q = tmalloc(16)) == NULL)
57     {
58       fprintf(stderr, "tmalloc failure\n");
59       exit(EXIT_FAILURE);
60     }
61
62   tfree(p);
63   tfree(q);
64
65   if ( (p = tmalloc(32)) == NULL)
66     {
67       fprintf(stderr, "tmalloc failure\n");
68       exit(EXIT_FAILURE);
69     }
70
71   obsd_strlcpy(p, arg, 32);
72
73   tfree(q);
74
75   return 0;
76 }
```

Double `tfree` example

After `tmalloc` call for q (line 56):

```
(gdb) x /20xw arena
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   0x00000000   0x00000000
0x8049de0 <arena+64>:   0x00000000   0x00000000   0x00000000   0x00000000
```



After `tfree` call for p (line 62):

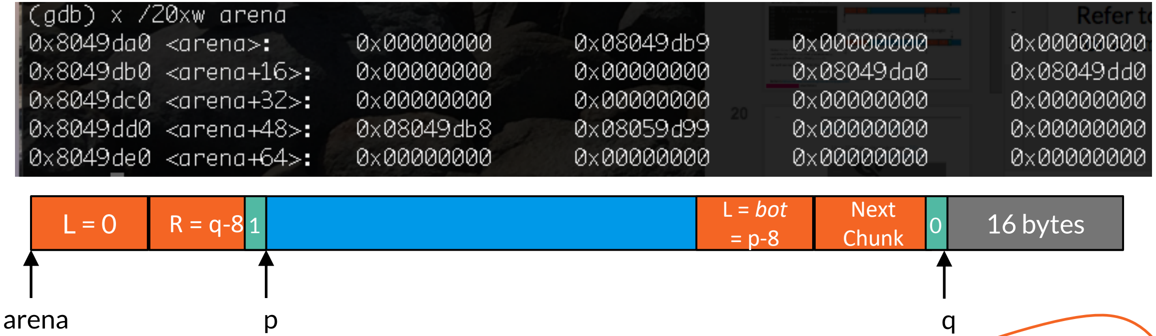
```
(gdb) x /20xw arena
0x8049da0 <arena>:      0x00000000   0x08049db9   0x00000000   0x00000000
0x8049db0 <arena+16>:   0x00000000   0x00000000   0x08049da0   0x08049dd0
0x8049dc0 <arena+32>:   0x00000000   0x00000000   0x00000000   0x00000000
0x8049dd0 <arena+48>:   0x08049db8   0x08059d99   0x00000000   0x00000000
0x8049de0 <arena+64>:   0x00000000   0x00000000   0x00000000   0x00000000
```

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.

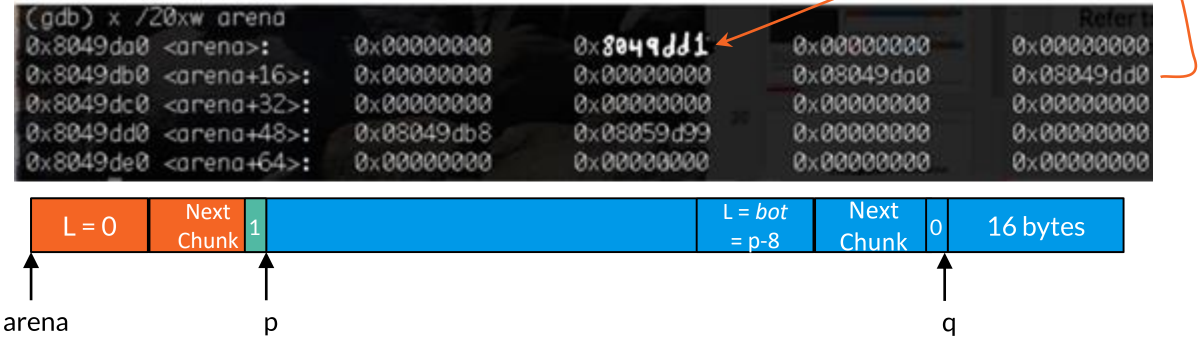
Double `tfree` example

```
46  int foo(char *arg)
47  {
48    char *p;
49    char *q;
50
51    if ( (p = tmalloc(16)) == NULL)
52      {
53        fprintf(stderr, "tmalloc failure\n");
54        exit(EXIT_FAILURE);
55      }
56    if ( (q = tmalloc(16)) == NULL)
57      {
58        fprintf(stderr, "tmalloc failure\n");
59        exit(EXIT_FAILURE);
60      }
61
62    tfree(p);
63    tfree(q);
64
65    if ( (p = tmalloc(32)) == NULL)
66      {
67        fprintf(stderr, "tmalloc failure\n");
68        exit(EXIT_FAILURE);
69      }
70
71    obsd_strlcpy(p, arg, 32);
72
73    tfree(q);
74
75    return 0;
76  }
```

After `tfree` call for p (line 62):



After `tfree` call for q (line 63):



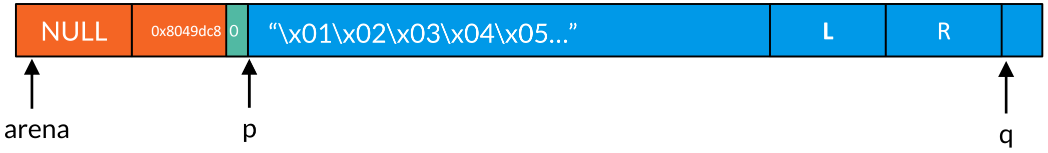
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

```c
int foo(char *arg)
{
  char *p;
  char *q;

  if ( (p = tmalloc(16)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (q = tmalloc(16)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(p);
  tfree(q);

  if ( (p = tmalloc(32)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(p, arg, 32);

  tfree(q);

  return 0;
}
```

Double `tfree` example

Our input buffer contains: \x01\x02\x03\x04\x05…\x11\x12\x13
After copying the buffer to the new p:

```
(gdb) x /20xw arena
0x8049da0 <arena>:      0x00000000   0x08049dc8   0x04030201   0x08070605
0x8049db0 <arena+16>:   0x0c0b0a09   0x100f0e0d   0x00131211   0x08049dd0
0x8049dc0 <arena+32>:   0x00000000   0x00000000   0x08049da0   0x08059d99
0x8049dd0 <arena+48>:   0x08049da0   0x08059d99   0x00000000   0x00000000
0x8049de0 <arena+64>:   0x00000000   0x00000000   0x00000000   0x00000000
```



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

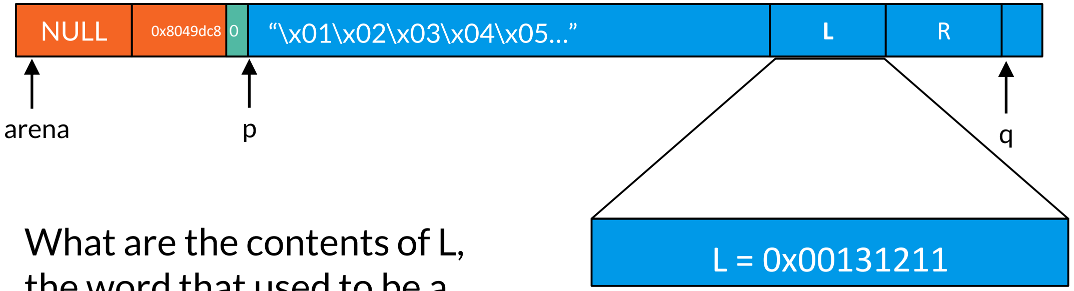
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation and to https://gitlab.cs.washington.edu/snippets/44 for the code used to generate these examples.

```c
int foo(char *arg)
{
  char *p;
  char *q;

  if ( (p = tmalloc(16)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (q = tmalloc(16)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(p);
  tfree(q);

  if ( (p = tmalloc(32)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(p, arg, 32);

  tfree(q);

  return 0;
}
```

Double `tfree` example

Our input buffer contains: \x01\x02\x03\x04\x05…\x11\x12\x13
After copying the buffer to the new p:

```
(gdb) x /20xw arena
0x8049da0 <arena>:     0x00000000  0x08049dc8  0x04030201  0x08070605
0x8049db0 <arena+16>:  0x0c0b0a09  0x100f0e0d  0x00131211  0x08049dd0
0x8049dc0 <arena+32>:  0x00000000  0x00000000  0x08049da0  0x08059d99
0x8049dd0 <arena+48>:  0x08049da0  0x08059d99  0x00000000  0x00000000
0x8049de0 <arena+64>:  0x00000000  0x00000000  0x00000000  0x00000000
```



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

Exploit hint 1: We can control the value stored at `q->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

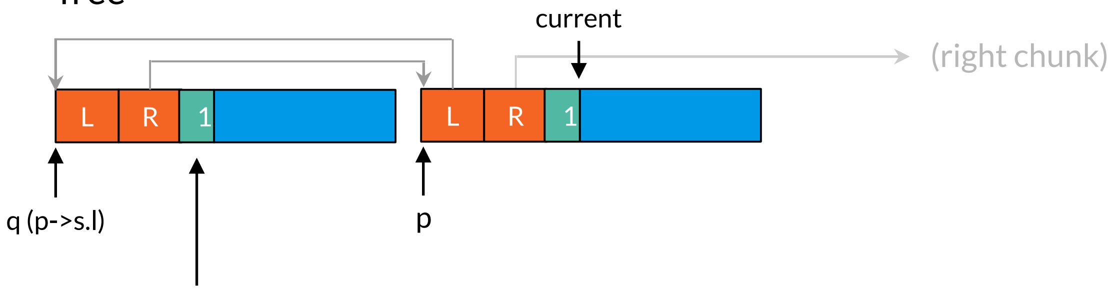
What would happen in `tfree(q)`?

```
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        }
```

Note: `tfree()` flips the naming in the variables (ie. `tfree(q)` renames the variable `q` from `foo()` to `p`, and `p` from `foo()` is referred to as `q` (when we set `q = p->s.l`).

Since this is confusing, we'll use `current` to refer to the `q` in `foo()`, and `p` and `q` to refer to the code in `tfree()`

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).

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

Double `tfree` example

What would happen in `tfree(q)`?

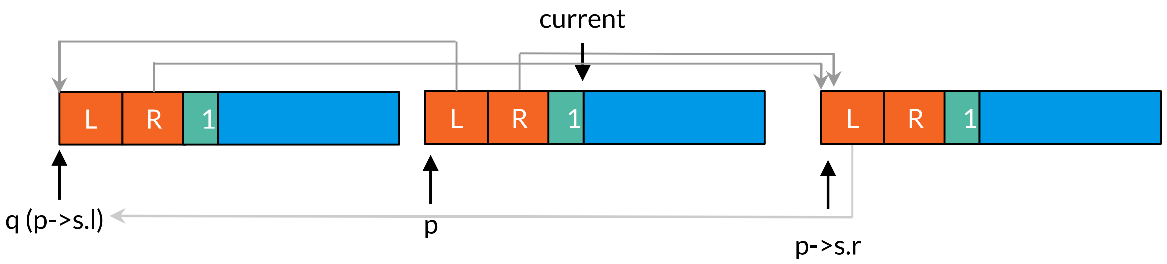
```
108     q = p->s.l;
109     if (q != NULL && GET_FREEBIT(q))
110       {
111         CLR_FREEBIT(q);
112         q->s.r       = p->s.r;
113         p->s.r->s.l = q;
114         SET_FREEBIT(q);
115         p = q;
116       }
```

Note: `tfree()` flips the naming in the variables (ie. `tfree(q)` renames the variable `q` from `foo()` to `p`, and `p` from `foo()` is referred to as `q` (when we set `q = p->s.l`).

Since this is confusing, we'll use `current` to refer to the `q` in `foo()`, and `p` and `q` to refer to the code in `tfree()`

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

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 (`q`, which `tfree` sets by reading `p.l`).

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

Final Words

- Good luck finishing lab 1a!

- Post questions on discussion board

next section: notes/hints for sploits 5-7, modular arithmetic, and 2DES