

# Recitation 11: More Malloc Lab

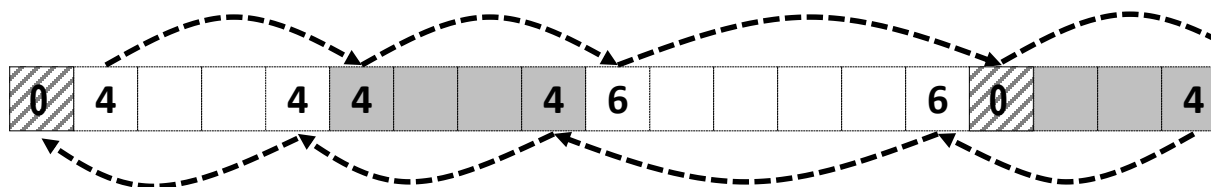
Instructor: TA(s)

# Understanding Your Code

- Sketch out the heap
- Add Instrumentation
- Use tools

# Sketch out the Heap

- Start with a heap, in this case implicit list



- Now try something, in this case, `extend_heap`

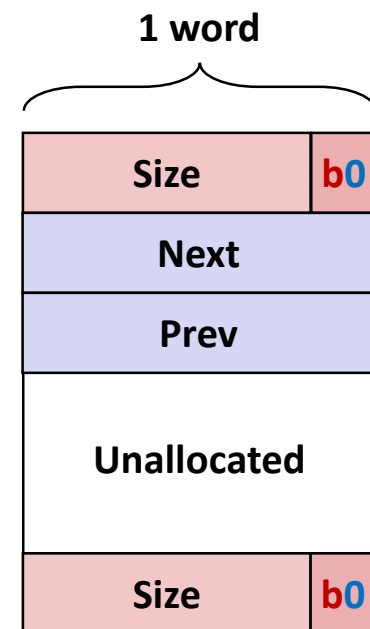
```

block_t *block = payload_to_header(bp);
write_header(block, size, false);
write_footer(block, size, false);
// Create new epilogue header
block_t *block_next = find_next(block);
write_header(block_next, 0, true);

```

# Sketch out the Heap

- **Here is a free block based on lectures 19 and 20**
  - Explicit pointers (will be well-defined see writeup and Piazza)
  - Optional boundary tags
- **If you make changes to your design beyond this**
  - Draw it out.
  - If you have bugs, pictures can help the staff help you



**Free  
Block**

# Use unions, avoid pointer arithmetic

## ■ Members of a union share storage space

- Only as big as necessary to store the largest member
- Compare with structs

```
struct S {  
    int i;  
    float f;  
}
```

```
union U {  
    int i;  
    float f;  
}
```

```
struct S s;  
s.i = 213;  
s.f = .213; // what is s.i?
```

```
union U u;  
u.i = 213;  
u.f = .213; // what is u.i?
```

## ■ At any time, writing to one member invalidates all the other members

- Useful when you need to use only one member at a time
- How can you use unions to define your block struct?

# Add Instrumentation

- **Remember that measurements inform insights.**
  - Add temporary code to understand aspects of malloc
  - Code can violate style rules or 128 byte limits, because it is temporary
  
- **Particularly important to develop insights into performance before making changes**
  - What is expensive throughput-wise?
  - How much might a change benefit utilization?

# Add Instrumentation example

- Searching in `find_fit` is often the slowest step
- How efficient is your code? How might you know?
  - Compute the ratio of blocks viewed to calls

```
static block_t *find_fit(size_t asize)
{
    block_t *block; call_count++;
    for (block = heap_listp; get_size(block) > 0;
         block = find_next(block))
    {
        block_count++;
        if (!(get_alloc(block)) && (asize <= get_size(block)))
        {
            return block;
        }
    }
    return NULL; // no fit found
}
```

# Add Instrumentation cont.

- **What size of requests?**
  - How many 8 bytes or less?
  - How many 16 bytes or less?
  - What other sizes?
  
- **What else could you measure? Why?**
  
- **Remember that although the system's performance varies**
  - The mdriver's traces are deterministic
  - Measured results should not change between runs



# Use tools

## ■ Use `mm_checkheap()`

- Write it if you haven't done so already
- Add new invariants when you add new features
- Know how to use the heap checker.
  - Why do you need a heap checker? 2 reasons.

## ■ Use `gdb`

- You can call `print` or `mm_checkheap` whenever you want in `gdb`. No need to add a while lot of `printf`'s.
- Offers useful information whenever you crash, like `backtrace`.

# mdriver-emulate

- **Testing for 64-bit address space**
- **Use correctly sized masks, constants, and other variables**
- **Be careful about subtraction between size types (may result in underflow/overflow)**
- **Reinitialize your pointers in mm\_init**

# Garbled Bytes

- **Malloc library returns a block**
  - mdriver writes bytes into payload (using memcpy)
  - mdriver will check that those bytes are still present
  - If malloc library has overwritten any bytes, then report garbled bytes
    - Also checks for other kinds of bugs
  
- **Now what?**
  
- **The mm\_checkheap call is catching it right?**
- **If not, we want to find the garbled address and watch it**

# Garbled Bytes and gdb

- Form pairs and get out a laptop
- Login to shark machine
- `wget http://www.cs.cmu.edu/~213/activities/rec11b.tar`
- `tar xf rec11b.tar`
- **mm.c is a fake explicit list implementation.**
  - Source code is based on mm\_baseline.c
  - A few lines of code are added that vaguely resembles what an explicit list implementation could have.

# GDB Exercise

■ `gdb --args ./mdriver -c ./traces/syn-array-short.rep -D`

```
(gdb) r
```

```
// Sample output follows
```

```
Throughput targets: min=6528, max=11750, benchmark=13056
```

```
Malloc size 9904 on address 0x800000010.
```

```
...
```

```
ERROR [trace ././traces/syn-array-short.rep, line 12]:
```

```
block 0 has 8 garbled bytes, starting at byte 0
```

```
...
```

```
Terminated with 2 errors
```

```
[Inferior 1 (process 13470) exited normally]
```

```
(gdb)
```

# GDB Exercise cont.

- What is the first address that was garbled?
  - Use gdb watch to find out when / what garbled it.

```
(gdb) watch * 0x800000010
```

```
(gdb) run
```

```
// Keep continuing through the breaks:
```

```
// mm_init()
```

```
// 4 x memcpy
```

```
Hardware watchpoint 1: *0x800000010
```

We just broke in  
after overwriting



```
Old value = -7350814
```

```
New value = 9928
```

```
mm_malloc (size=50084) at mm.c:272
```

## Second Exercise

Well fine, the bug from the first exercise was very artificial.  
No one just sets bytes to 0 for no reason.

Try this more plausible exercise:

```
$ gdb --args ./mdriver-2 -c traces/syn-array-short.rep
```

What error was printed to the console?

The function that prints the error is named `malloc_error`. Add a breakpoint for it if you want.

## Second Exercise

**The library must've written the header and footer for the out-of-bounds payload at some point. Add a watchpoint for either address, or both.**

**...So, the writes occurred in `place`. Is the `place` function wrong, or was it just given a bad argument?**

**Hint: the bug is found in at basically the same place as last recitation's bug.**

**It's caused by a careless typo, like nearly all others bugs.**



# Tips for using our tools

- Run `mdriver` with the `-D` option to detect garbled bytes as early as possible. Run it with `-V` to find out which trace caused the error.
- Note that sometimes, you get the error within the first few allocations. If so, you could set a breakpoint for `mm_malloc` / `mm_free` and step through every line.
- Print out local variables and convince yourself that they have the right values.
- For `mdriver-emulate`, you can still read memory from the simulated 64-bit address space using `hprobe` instead of `x` / `gx`.

# MallocLab

- Checkpoint due Tuesday (**tomorrow**)
- 7% of final grade (+ 4% for checkpoint)
- Read the writeup. It even has a list of tips on how to improve memory utilization.
- Rubber duck method
  - If you explain to a rubber duck / TA what your function does step-by-step, while occasionally stopping to explain why you need each of those steps, you'd may very well find the bug in the middle of your explanation.
  - Remember the “debug thought process” slide from Recitation 10?

# Autolab throughput issue

- **Autolab is giving very different throughputs than what you will see from driver.pl on shark machines**
- **If Autolab does not work until the deadline, we will run your submissions manually on the shark machines to assign a score**
  - That being said, use the score given driver.pl as a reference to your final score
  - We will run every submission a few times and take the maximum score