This document contains the questions and solutions to the final exam given in Spring 2015. The class was taught by Julie Zelenski and Michael Chang. This was a 3-hour exam.

Final questions

Problem 1: x86-64 assembly
Below is the assembly code generated for the **pinky** function.

```
pinky:
       push   %rbp
       push   %rbx
       sub    $0x18,%rsp
       mov    %rsi,%rbx
       mov    %edx,%ebp
       callq  <strlen>
       mov    %eax,0xc(%rsp)
       callq  <rand>
       jmp    .L2
.L1:
       sub    %eax,(%rbx)
       mov    0xc(%rsp),%ecx
       lea    0xe(%rcx,%rcx,4),%ecx
       mov    %ecx,0xc(%rsp)
       add    $0x4,%eax
.L2:
       test   %eax,%eax
       js     .L1
       mov    %ebp,%edx
       imul   0x4(%rbx),%edx
       lea    0xc(%rsp),%rsi
       mov    $0x0,%edi
       callq  <pinky>
       neg    %eax
       add    $0x18,%rsp
       pop    %rbx
       pop    %rbp
       retq
```

Fill in the blanks in the C code below to match the assembly above. Note this is nonsense code, not intended to do anything at all reasonable. Your code should not contain any typecasts.

```
int pinky(char *param1, int *param2, int param3)
{

    int local = ____________________________ ;

    for ( int i  = ___________________ ; ____________________ ; ___________________ ) {

        ____________________________________________________________________;
```

```
        ____________________________________________________________;
    }

    ____________________________________________________________;
}
```

Problem 2: Symbol tables

The symbol table section of an ELF file is a list of text and data symbols, including the name, address, and size of each symbol. The **`strip`** utility removes the symbol table section. A library file is an ELF file consisting of one or more **`.o`** files aggregated into an archive.

What purpose does the symbol table serve for a library file? If you strip a library file, can it still be used? Why or why not?

What purpose does the symbol table serve for an executable file? If you strip an executable, can it still be used? Why or why not?

The namelist (**`nm`**) output for your binary bomb executable includes these function symbols:

```
0000401893    25  T   explode_bomb
00004023a0    38  T   defuse_bomb
```

Each of these two functions does what its name implies. You want to know if the functions have the same prototype. Can you determine this from the information in the symbol table? Explain.

You edit the symbol table of your binary bomb executable and overwrite the address for **`explode_bomb`** with the address of **`defuse_bomb`** in an attempt to convert explosions into defusions. You run your modified executable under gdb and set a breakpoint:

```
myth> gdb bomb
Reading symbols from bomb...done.
(gdb) break explode_bomb
Breakpoint 1 at 0x4023a0
(gdb) run
Starting program: /Users/zelenski/bomb
...
KABOOM!! The bomb has blown up.
```

Your breakpoint is not hit, yet the bomb has exploded. Explain what happened.

Outline a different technique for hacking your binary bomb executable that would successfully substitute **`defuse_bomb`** for **`explode_bomb`**.

Problem 3: Runtime stack

The `nonsense` function declares a variable-length stack array. Its C source is below on the left and its generated assembly is on the right.

```
size_t nonsense(size_t n)
{
    char buf[n];
    buf[n-1] = '\0';
    return sizeof(buf);
}
```

```
push   %rbp
mov    %rsp,%rbp
mov    %rdi,%rax
lea    0xf(%rdi),%rdx
and    $0xfffffffffffffff0,%rdx
sub    %rdx,%rsp
movb   $0x0,-0x1(%rsp,%rdi,1)
mov    %rbp,%rsp
pop    %rbp
retq
```

A program contains a call to `nonsense(6)`. Before executing the instruction `callq <nonsense>`, the value of `%rsp` is `0x7ffff060`. Trace the execution of `nonsense` and answer these questions.

What is the value of `%rsp` when executing the underlined `movb` instruction?

What value is returned by the call `nonsense(6)`? If you believe the result is unpredictable or the call will crash, explain why.

The `strange` function is very similar to `nonsense`. Below is its C code and generated assembly.

```
size_t strange(size_t n)
{
    char buf[n];
    buf[n-1] = '\0';
    return strlen(buf);
}
```

```
push   %rbp
mov    %rsp,%rbp
lea    0xf(%rdi),%rax
and    $0xfffffffffffffff0,%rax
sub    %rax,%rsp
movb   $0x0,-0x1(%rsp,%rdi,1)
mov    %rsp,%rdi
callq  <strlen>
mov    %rbp,%rsp
pop    %rbp
retq
```

What value is returned by the call `strange(6)`? If you believe the result is unpredictable or the call will crash, explain why.

For all sufficiently large values of `n`, a call to `strange(n)` crashes during execution, yet a call to `nonsense(n)` executes without incident. Explain why.

If `strange` were changed to return `rand()` instead of `strlen(buf),` would a call to `strange(n)` still crash for all large values of `n`? Explain why or why not.

The **offbyone** function adds a bug to **strange**. Below is its C code and generated assembly.

```
size_t offbyone(size_t n)
{
    char buf[n];
    buf[n] = '\0';
    return strlen(buf);
}
```

```
push   %rbp
mov    %rsp,%rbp
lea    0xf(%rdi),%rax
and    $0xfffffffffffffff0,%rax
sub    %rax,%rsp
movb   $0x0,(%rsp,%rdi,1)
mov    %rsp,%rdi
callq  <strlen>
mov    %rbp,%rsp
pop    %rbp
retq
```

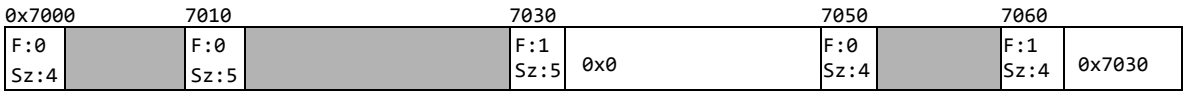
The function intends to zero the last element, but an off-by-one error causes it to write one past the end. The bug is real, but largely asymptomatic. Why? Identify the exact conditions under which a call to **offbyone** overwrites something it should not and indicate precisely what data is overwritten.

Problem 4: Heap allocator

You are writing code for a simple allocator that uses a block header, no footer, and maintains an explicit free list. Implementation details of this allocator include:

- The total block size (size of payload + size of header) for all blocks is required to be an exact power of 2. The payload must be at least 8 bytes and block header is 1 byte, so the minimum total block size rounds up to 16. Payload pointers are not required to be aligned.
- The block header is an 8-bit unsigned char that bit-mashes together the block information:
  - most significant bit is 1 if block is freed, 0 if in-use
  - lower 7 bits store $\log_2$ of the total block size
- The allocator maintains an explicit free list as a singly-linked list stored in the block payload. A global variable points to the header of the first free block (or NULL if the free list is empty). The first 8 bytes of the payload of a free block store a pointer to the header of another free block. The last free block on the list stores NULL in its payload.

Here is an example heap after a few requests have been serviced:



This heap starts at address 0x7000 and has size 112 bytes. Three blocks are in-use blocks, two are free. The in-use payloads are shown shaded in gray. The header of the first block has the most significant bit off (block is in-use) and the lower bits store 4 (total block size is 16 or 2^4). The free list points to the header at 0x7060, the payload of that block points to the header at 0x7030, the payload of that block stores NULL.

Below are the allocator's global variables, constants, and type definitions.

```
// type: header stores per-block housekeeping
typedef unsigned char Header;

static Header *heap_start;    // first header in heap segment
static size_t  heap_size;     // number of bytes in heap segment
static Header *free_list;     // header of first block on free list (NULL if none)

// masks to divide header into most significant bit and all other bits
#define MSB_MASK   (1 << ((sizeof(Header)<< 3) -1))
#define SIZE_MASK  (MSB_MASK - 1)
```

The **MSB_MASK** and **SIZE_MASK** are used to extract the most significant bit from the other bits in the header. You run the preprocessor and see that it replicates these complex expressions at each use and are concerned this will degrade throughput. Never fear, gcc has your back! What specific compiler optimization will be applied to avoid repeatedly re-computing the mask?

The **is_free** function is given a pointer to a block's header and returns true if the block is free, false otherwise.

```
bool is_free(Header *hdr)
{
   return  /* expr */ ;
}
```

is_free is missing the test expression. From the choices below, circle all options that evaluate to the correct result.

```
((*hdr & MSB_MASK) == MSB_MASK)

(*hdr & MSB_MASK)

~(*hdr ^ MSB_MASK)

*hdr < 0

*(signed char *)hdr < 0

*(int *)hdr < 0

(int)*hdr < 0
```

Implement **get_blocksize**. Given a pointer to a block's header, the function returns the total number of bytes in the block.

```
size_t get_blocksize(Header *hdr)
{
```

Implement **get_neighbor**. Given a pointer to a block's header, the function returns a pointer to the header of the neighboring block to the right, i.e., at next higher address in heap. If this block is lastmost in the heap, the function returns **NULL**.

```
Header *get_neighbor(Header *hdr)
{
```

The first 8 bytes of the payload of a free block are used to store a pointer to the header of the next block on the free list. Helper functions **get_next** and **set_next** read and write those links.

```
Header *get_next(Header *hdr);
void set_next(Header *first, Header *second);
```

Implement the **set_next** function to link from first to second using a single call to **memcpy**.

```
void set_next(Header *first, Header *second)
{

   memcpy(____________________ , ________________________ , ____________________);
}
```

On second thought, you recall the lab8 performance timings and realize there has to be a better way. Re-implement **set_next** to link from first to second using a single pointer assignment.

```
void set_next(Header *first, Header *second)
{
```

You will use **set_next** and **get_next** in implementing **remove_from_freelist** and can assume both functions work correctly.

The function **remove_from_freelist** is given a pointer to a header currently on the free list. An implementation is started below that searches for the entry on the list. Add the necessary code to then remove the entry from the list.

```
void remove_from_freelist(Header *to_remove)
{
    Header *prev = NULL, *cur = free_list;
    while (cur != to_remove) {
        prev = cur;
        cur = get_next(cur);
    }
```

In this allocator, two adjacent blocks can only be coalesced if they are both free and have the same total block size. Why must they be the same size?

Implement the **myfree** function. It should mark the block as free and add it to front of the free list. If the neighboring block to the right is free and same size, it should be coalesced into this one and the neighbor's header is removed from the free list. Attempt to coalesce only a single neighbor, not a sequence of neighbors. You may make use of any of the previous helper functions and can assume they work correctly.

```
void myfree(void *payload)
{
```

This allocator maintains the free list in a singly-linked list and profiling shows traversing the list to be a performance bottleneck for both **`mymalloc`** (which traverses to find an appropriately-sized block) and **`myfree (`**which traverses within **`remove_from_freelist`** when coalescing). Your partner proposes two possible strategies to combat this: double-linking the list or segregating the list by size. Consider each proposed change and provide your cost/benefit evaluation of pursuing it. Your analysis should address the specific impacts of:

What is the added code complexity?
What is the expected effect on throughput of **`mymalloc`**? on throughput of **`myfree`**?
What is the change in utilization?

Strategy A: double-link the free list.

Strategy B: segregate the free list by size into multiple lists.

Solutions

Problem 1: x86-64 assembly

```
int pinky(char *param1, int *param2, int param3)
{
    int local = strlen(param1);
    for (int i = rand(); i < 0 ; i += 4) {
      *param2 -= i;
      local = 5*local + 14;
    }
    return -pinky(NULL, &local, param2[1]*param3);
}
```

Problem 2: Symbol tables

The linker reads the library's symbol table to resolve symbol references during linking. Stripped of this road map, the library is a useless bag of bytes.

The executable's symbol table identifies symbols by name/address. Without it, executable runs just fine, but is less friendly to introspection by debugger/valgrind.

No, the symbol table contains no type information.

References were resolved to their final addresses when the executable was linked, and **`callq`** instructions were hard-wired to absolute locations. Changing the address in the symbol table has no consequence to the control flow: what was compiled as a call to **`explode_bomb`** remains a call to **`explode_bomb`**. However, gdb uses the symbol table to map names to addresses, thus the breakpoint set on **`explode_bomb`** was actually set on the address for **`defuse_bomb`**.

Here are some options. Each works by editing the binary-encoded machine instructions within the **`.text`** section of the ELF executable:

- overwrite the instructions for **`explode_bomb`** with a copy of instructions from **`defuse_bomb`** (a little bit problematic as defuse requires more bytes according to namelist output, but ok to ignore that detail)
- replace the first instruction of **`explode_bomb`** to redirect to **`defuse_bomb`** (unconditional jmp, callq and ret)
- find every **`call`** instruction in **`.text`** section that has **`explode_bomb`** as the target and change target address to **`defuse_bomb`** (harder than it sounds, as call instructions are all pc-relative, but you get the idea)

Problem 3: Runtime stack

`%rsp = 0x7ffff040`

`nonsense(6)` returns **`6`**

`strange(6)` returns a number between 0 and 5 (which value depends on whether uninitialized contents of **`buf`** happen to include a null byte).

The stack segment has a maximum size. Adjusting stack point to create space for an array larger than segment size will set **`%rsp`** to an invalid address. This puts us in a precarious position, but

from there `nonsense(huge)` doesn't access the low part of that array or further extend the stack, it simply cleans up the space and returns, so it escapes unscathed. When the call to `strange(huge)` reaches that same point, it invokes `strlen` which reads/writes from (invalid) top of stack—boom!

Yes. The `callq` instruction fails when trying to push the return address.

A call to `offbyone(n)` for any `n` that is a multiple of 16 will zero the low byte of `%rbp` register value that was saved on stack. When the function returns, the value that caller previously had stored in `%rbp` will be corrupted.

Problem 4: Heap allocator

Constant folding. If an expression only involves constants, the compiler evaluates it at compile-time and will use the immediate value in place of generating the instructions to compute it again.

These three work:

```
((*hdr & MSB_MASK) == MSB_MASK)

(*hdr & MSB_MASK)

*(signed char *)hdr < 0
```

```
size_t get_blocksize(Header *hdr)
{
    return 1L << (*hdr & SIZE_MASK);
}

Header *get_neighbor(Header *hdr)
{
    Header *next = hdr + get_blocksize(hdr);
    if (next >= heap_start + heap_size) return NULL;
    return next;
}

memcpy(first + 1, &second, sizeof(void *));

*(Header **)(first + 1) = second;

Header *next = get_next(cur);
if (prev == NULL)
         free_list = next;
    else
         set_next(prev, next);
```

Block sizes have to be exact power of 2.

```
void my_free(void *ptr)
{
    Header *hdr = (Header *)ptr - 1;
    *hdr |= MSB_MASK;
```

```
    set_next(hdr, free_list);
    free_list = hdr;

    Header *next = get_neighbor(cur);
    if (next && *hdr == *next) { // compares size and freed
        remove_from_freelist(next);
        *hdr += 1;
    }
}
```

Double-link:
Code changes: Must splice backward link in addition to forward when adding/removing from list
Throughput: mymalloc unchanged, myfree was O(N) now O(1)
Utilization: min block size increases to 32 (min payload >= 16 bytes for two pointers)

Segregate:
Code changes: segregate by powers of two, use log size from header bits as index into array of linked lists, same code to add/remove from list
Throughput: mymalloc was O(N) now O(1) (takes first block), myfree was O(N) now O(N/B) (number of buckets ~32)
Utilization: mymalloc now operating as best-fit, may result in less fragmentation