# 6.858 Quiz I Review 2018

Cattalyya Nuengsigkapian
Mar 19, 2018

## Baggy Bound

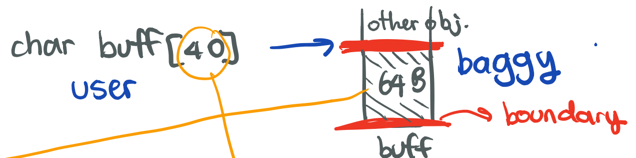
- Bound Checking to avoid buffer overflow

Goals: Space, Time Efficient, Backward Compatibility

### ♡ Components

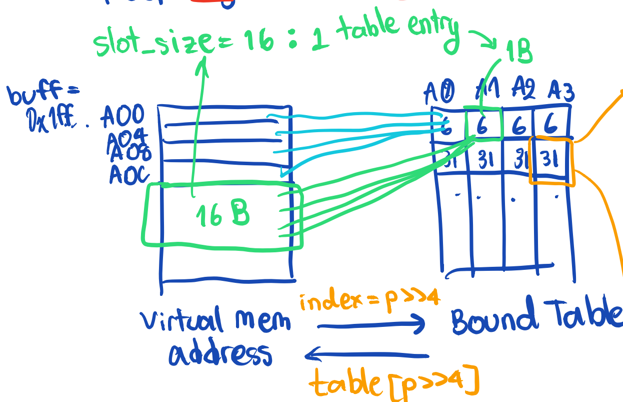
#### ① Buddy Allocator

- make sure allocated size = $2^k$
  - allocated size: of the whole obj. ex. struct (NOT inner components) and whole array
  - $2^k$: user can use this much space for this obj.
- minimum size ≥ slot_size
- no two objs share the same slot




#### ② Bound Table

- Keep "log" of the "allocated size" * NOT obj. size




Initialize table slot with 31 (base = 0, size = $2^{31}$) for Backward Compatibility
↓
pointer to obj. allocated by external uninstrumented library (unsupported) will NOT raise issue when baggy bound code try to access

→ Deallocate also set the slot to 31
- mark available

Ex. buff[28] >>4 ⟶ table[A1] = 6
= buff + 0x1C
= 0x1ff...A1C

| size = table[p>>4] (4 = lg(slot_size)) |
|---|
| Base = p & ~(size−1) |

Base: clear low bits of p

### ♡ Checking

Bound checking on p' pointer derived from pointer p

| p ^ p' >> table[p>>4] |
|---|

Ex. p' = p + 30

#### ① Arithmetic

- In bound OK
- over bound
  - exceed ≤ $\frac{slot_size}{2}$ ⟶ set OOB bit = most significant bit
    - out of bound → *p → ERROR bc. trying to access an inaccessible page (high memory start with 1)
  - o.w. ⟶ raise an ERROR = terminate program

#### ② Dereference

same example

buff[65] = 'A' }
*(buff + 70) = 'A' } ERROR
buff[45] = 'A' work fine

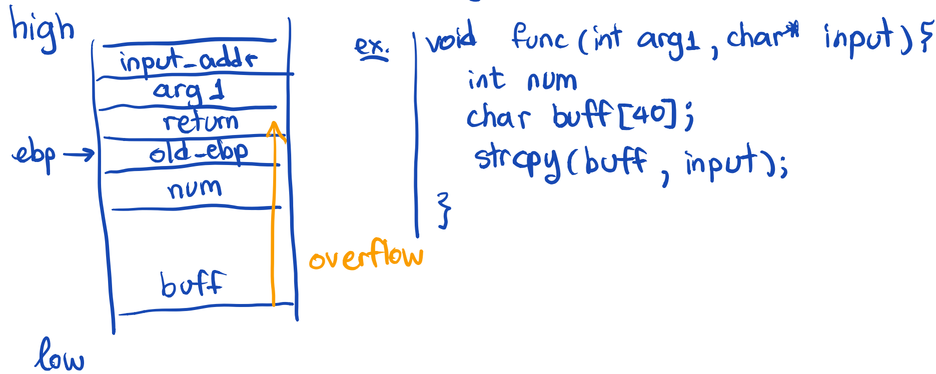
baggy bound only keep track of allocated size
⇒ Don't know user requested size

# Buffer Overflow + Lab1

Cattalyya N.

(doesn't have much time to cover)

Please make sure you understand what you are doing in Lab 1




**Attack** to change control flow

- overwrite return address
- " function pointer
- change data value → could force to run some path

Ex.

```
if (num = 3) {
    printf(secret);
}
```

**Some solutions**

**1 Stack Canaries**

- compiler mechanism inserting canary before return address on the stack

Canary Property
- random
- contain NULL termination
  to avoid attack writing further if he can guess canary
  bc. most function handling string will terminate
  ↳ gets()

**2 ASLR**

- random the address space so that attacker can't easily hardcode the address.

**3 NX**

= Non executable → can't execute code on the stack.

Review old quizes

Cattalyya N.

## 2017

**7. [6 points]:**

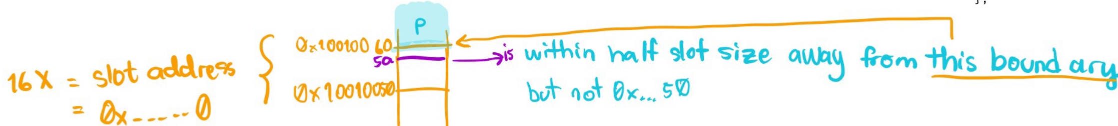
The program prints `0x9001005a`. What was $N$? Hint 1: there is only one correct answer. Hint 2: recall that Baggy manipulates certain bits in pointers for bookkeeping.

buff = 0b 1001 .... —original pointer→ 0b 0001 ... = 0x 1001005a

↳ OOB bit is set.

```
p     struct foo {
p+ 0x0   int a;   4
p+ 0x4   int b;   4
p+ 0x8   int c;   4
p+ 0xC   char d[12];  4:12B
         };  18
```
} 24B

0x 1001005a = (p + 0x C) + N    (p→d)




5

0x1001005a = 0x1001006C + N

N = -18 #

This question wasn't covered during exam review but many sudents want detailed solution.

## 2011

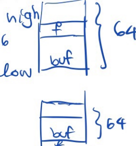
### IV Baggy bounds checking

Consider a system that runs the following code under the Baggy bounds checking system, as described in the paper by Akritidis et al, with slot_size=16:

(doesn't take care of bound inside struct)

```
1  struct sa {
2    char buf[32];
3    void (*f) (void);
4  };
5
6  struct sb {
7    void (*f) (void);
8    char buf[32];
9  };
10
11 void handle(void) {
12   printf("Hello.\n");
13 }
14
15 void buggy(char *buf, void (**f) (void)) {
16   *f = handle;
17   gets(buf);
18   (*f) ();
19 }
20
21 void test1(void) {
22   struct sa x;
23   buggy(x.buf, &x.f);
24 }
25
26 void test2(void) {
27   struct sb x;
28   buggy(x.buf, &x.f);
29 }
30
31 void test3(void) {
32   struct sb y;
33   struct sa x;
34   buggy(x.buf, &y.f);
35 }
36
37 void test4(void) {
38   struct sb x[2];
39   buggy(x[0].buf, &x[1].f);
40 }
```

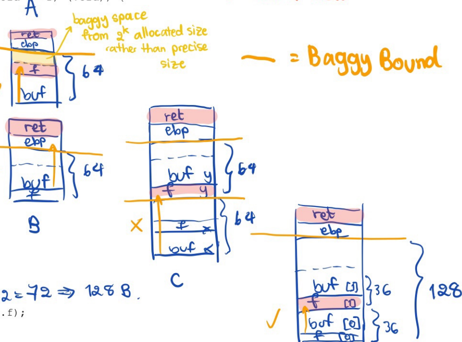
64B; 32+4=36




36×2=72 ⇒ 128 B.

○ = part that you can overwrite to change the control flow

— = Baggy Bound




Assume the compiler performs no optimizations and places variables on the stack in the order declared, the stack grows down (from high address to low address), that this is a 32-bit system, and that the address of `handle` contains no zero bytes.



**9. [6 points]:**

A. **True** / **False** If function `test1` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. (circled: True)

B. **True** / **False** If function `test2` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. (circled: False) * ~~Def~~ — prevent by Baggy Band

C. **True** / **False** If function `test3` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. (circled: False)

D. **True** / **False** If function `test4` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. (circled: True)

For the next four questions, determine what is the minimum number of bytes that an adversary has to provide as input to cause this program to likely crash, when running different test functions. Do not count the newline character that the adversary has to type in to signal the end of the line to `gets`. * Recall that `gets` terminates its string with a zero byte.

**10. [4 points]:** What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test1` to crash?

32 Overwrite f

**11. [4 points]:** What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test2` to crash?

64 − 4 = 60 } Baggy Bound

**12. [4 points]:** What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test3` to crash?

64

**13. [4 points]:** What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test4` to crash?

32 overwrite f