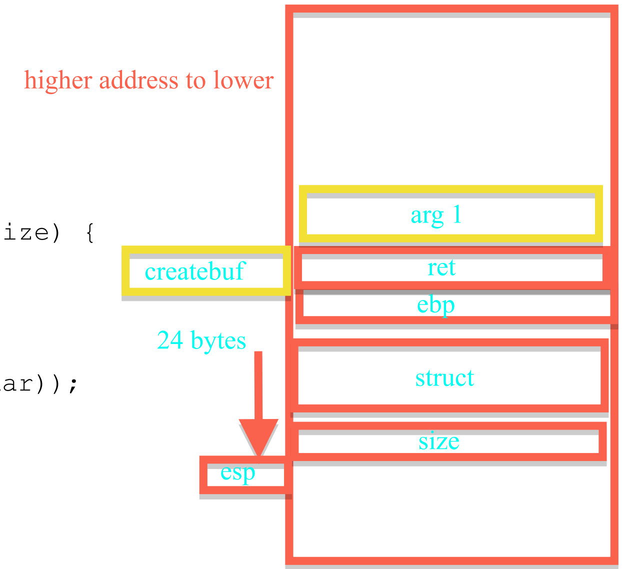
1. ***A Stack Walks into a Bar and Says "It's Hard to Maintain Discipline While Getting Smashed" (20 points):*** Consider the following C datatype, intended to provide some protection against buffer overflow:

```
struct safe_buffer {
  int size;
  char * buffer;
} mybuf;
```

12 buffer, then 4 from size



And the following function that creates a safe buffer:

```
void createbuf(struct safe_buffer *buf, int size) {
  int i;
  char tempbuf[12];

  (*buf).size=size;
  (*buf).buffer=calloc((*buf).size, sizeof(char));
  gets(tempbuf);
  for (i=0; i<size; i++)
    (*buf).buffer[i]=tempbuf[i];
  return;
}
```

Your friend argues that this function takes a size parameter and allocates that much buffer space – then only writes that much space to the buffer, ensuring that the safe buffer cannot overflow. Your friend is completely wrong. Prove that the code can be exploited – give us a string (plain text) that could be used to form an exploit string as you did in the buffer lab. You exploit string should maintain the correct value of the saved ebp on the stack, but should change the saved return address to *0x080485e8* so that the return from createbuf will take us to that address. Don't worry about the call to hex2string – just give us plain text. Here's some useful data from execution on IA32 Linux – the disassembled call to createbuf:

```
8048512:        e8 e2 fe ff ff          call   80483f9 <createbuf>
```

And the values of %esp and %ebp, and a gdb dump of some of the stack, after the call to gets in createbuf has completed and we are inside the for loop in createbuf:

```
esp             0xffffdb40
ebp             0xffffdb58
```

this is where the buffer starts

```
(gdb) x/32x 0xffffdb40
0xffffdb40:     0xffffdb48      0x00000001      0x74617257      0x00000068
0xffffdb50:     0x00000000      0x00000000      0xffffdbc8      0x08048517
0xffffdb60:     0x08049720      0x0000000a      0x00000000      0x00000000
0xffffdb70:     0xf63d4e2e      0x00000000      0x00000000      0x00000000
0xffffdb80:     0x00000000      0x00000000      0x00000000      0x08048340
0xffffdb90:     0x00000000      0x080496f4      0xffffdba8      0x080482a1
0xffffdba0:     0x00299ff4      0x00298204      0xffffdbd8      0x08048569
0xffffdbb0:     0x00183e25      0xffffdc6c      0xffffdbd8      0x00299ff4
```

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 (58 db ff ff) (e8 85 04 08)

2. ***Cache Me If You Can! (30 points):*** Consider the following C function.

```
                      4                    2                    2
void shrink(int *old, int *new, int dim_new, int shrink_factor) {
  int i, j;
  int u, v;

i=0,1 for (i=0; i<dim_new; i++) {
  j=0,1 for (j=0; j<dim_new; j++) {
    0,1,2,3 new[i*dim_new+j]=0;
      for (u=0; u<shrink_factor; u++) {
        for (v=0; v<shrink_factor; v++) {
          new[i*dim_new+j]+= 0,2,1,3            4      = 0,8,4,12
               old[(i*shrink_factor+u)*dim_new*shrink_factor
                    +(j*shrink_factor+v)];
        }                 0,2,1,3            old[0,2,1,3,8,10,9,11,4,6,5,7,12,14,1
      }                                      3,15] =
      new[i*dim_new+j]/=shrink_factor*shrink_factor;
    }     0,2,1,3                 4
  }
}
```

This function effectively takes a 2D matrix (`int *old`) and outputs a new 2D matrix based on this called (`int *new`). The parameter `dim_new` defines the size of the new 2D matrix – it is effectively a matrix of (`dim_new * dim_new`) integers. The last parameter, `shrink_factor`, is how many times smaller one dimension of the new matrix is relative to the old matrix. So if we went from a 400x400 matrix to a 100x100 matrix, `dim_new` would be 100 and `shrink_factor` would be 4. This could be used for something like image scaling. The technique to shrink the matrix will basically just use a simple, non-overlapping average – probably not good enough for high quality image scaling, but we'll do the best we can with it.

This problem is intended to be the most challenging one on this exam – so before continuing be sure you understand the original code first – it may be useful to run through an example of the shrinking on a small matrix – like a 4x4 matrix shrinking to a 2x2 matrix (scaling factor is 2).

We want to optimize this code by using strength reduction and common subexpression elimination on as many multiplies as possible, by eliminating unneeded memory references, and by using blocking to improve locality in the loop structure. There are lots of ways to attack this, but we are going to force you to finish the one we have started on the next page (this one cuts the runtime of shrink in half). The author of this code segment has followed a ***horrible*** coding practice of naming some of their variable names in a completely irrelevant way to the code function – so you cannot rely on the variable names to help you discern their functionality.

Your job is to fill in the blanks to make this code work correctly. The blanks we have inserted will look like this: ______**A**______ where the letter at the center of the blank is the label for the space on the answer key. So you should have 5 labels (**A-E**) to fill in for this problem. MIN(X,Y) is a macro that returns the minimum of values X and Y.

```
                                 4                     2            2
void shrink_fast(int *old, int *new, int dim_new, int shrink_factor)
{
  int i, j;
  int u, v;

  int iidim,jj,ii;

  // HINT - all labels should be filled with one of these names
  int platypus, kangaroo, echidna, cassowary, koala, dingo, wallaby,
      wombat;

  int dimshrink,sf2,sf2dim,bdim;
     4
  dimshrink=dim_new*shrink_factor;
4 sf2=shrink_factor*shrink_factor;
8 sf2dim=sf2*dim_new;
20 bdim=BSIZE*dim_new;

  iidim=0;                   2           4
  for (ii=0; ii<dim_new; ii+=BSIZE) 4 {
    for (jj=0; jj<dim 2 new; jj+=BSIZE) {
      wallaby=MIN(ii+BSIZE,dim_new);
  0   wombat=iidim;
  4x  platypus=iidim*sf2;
 0,2  cassowary=jj*shrink_factor;
      for (i = ii; i < _wallaby A____; i++){
        kangaroo=__cassowary B__+sf2dim;
        dingo=MIN(jj+BSIZE,dim_new);
 0,2    echidna=cassowary;
        for (j = jj; j <_dingo C____; j++){
          koala=0;                                     4
          for (u=platypus; u<kangaroo; u+=dimshrink) {
            for (v=echidna; v<echidna+shrink_factor; v++) {
             koala    D____+=old[u+v];     2x+2
            }
          }
          new[_wombat E____+j]=koala/sf2;
          echidna+=shrink_factor;
        }
        wombat+=dim_new;
        platypus+=sf2dim;
      }
    }
    iidim+=bdim;
  }
}
```