Computer Science 5271
Fall 2013
Midterm exam (solutions)
October 14th, 2013
Time Limit: 75 Minutes, 1:00pm-2:15pm

- This solution set contains 11 pages (including this cover page) and 5 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic device: no calculators, smart phones, laptops, etc. You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember, we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- By signing below you certify that you agree to follow the rules of the exam, and that the answers on this exam are your own work only.

The exam will end promptly at 2:15pm. Good luck!

Your name (print): _____

Your UMN X.500 (AKA CSELabs login, email): _____`@umn.edu`

Number of rows ahead of you: _____ Number of seats to your left: _____

Sign and date: _____

Question	Points	Score
1	15	
2	20	
3	20	
4	25	
5	20	
Total:	100	

1. Your consulting firm Tesseract Security Solutions had been hired to configure a multilevel-secure classification and access control system for the intelligence agency of a small, landlocked nation. You and a coworker are designing a lattice model for the project. The nation has three levels of classification, named "Unclassified", "Secret", and "Top Secret" and abbreviated U, S, and TS respectively. There are also three specialized compartments for clandestine projects codenamed APPLEBUTTER, BLINDSIDE, and CRAYOLA (abbreviated A, B, and C). A co-worker got most of the way through drawing a diagram of the lattice, but then some of his work was lost when he spilled coffee on the paper. (The diagram is on the next page, and rotated 90 degrees to fit on the page.)

   (a) (5 points) Fill in the correct labels on the five points in the lattice which are missing labels (shown with a dashed outline).

   (b) (5 points) Carol is an intelligence analyst cleared to the Secret level, with access to the APPLEBUTTER and CRAYOLA compartments; the corresponding point in the lattice is shown with a bold outline. Label with an "R" all the points in the lattice from which Carol can read information under a BLP policy, and label with a "W" all the points she can write to.

   (c) (5 points) The intelligence agency is considering starting five new projects to be codenamed DANCEHALL, EXCELSIOR, FACEPAINT, GALOSHES, and HEXDUMP. If you add new compartments for these new projects in addition to the existing ones, how many total points will be in the resulting lattice?

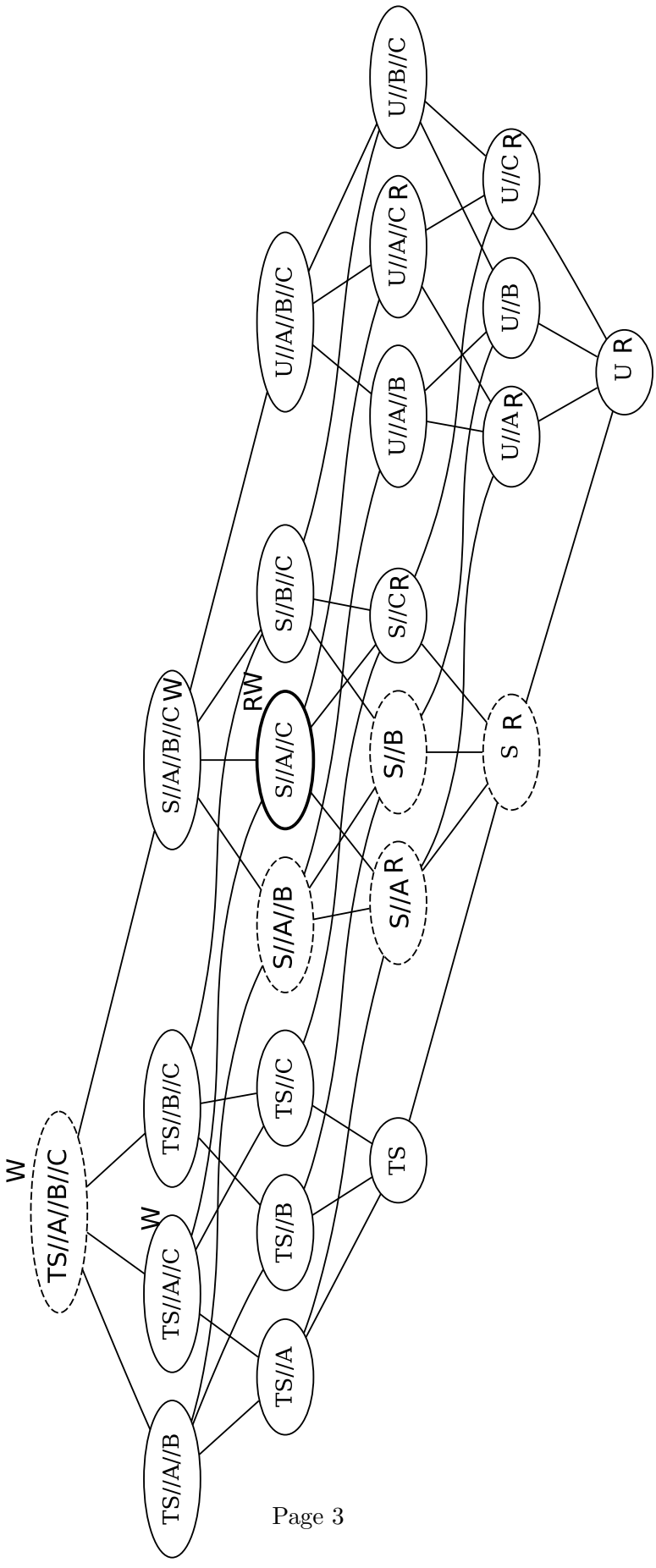
Solution: $3 \cdot 2^{3+5} = 3 \cdot 2^8 = 3 \cdot 256 = 768$

There are three levels, and the number of points on each level is equal to the number of subsets of the set $\{A, B, C, D, E, F, G, H\}$.

In the old lattice, there are 8 points on each level. If you hadn't noticed that $8 = 2^3$, consider what happens when you add DANCEHALL: for every point in the old lattice, there is a copy of that point in the new lattice, and also a copy of that point with //D added. Continuing this pattern the number of points in the lattice doubles with every new compartment added.

It's also possible, though more cumbersome, to count the points based on how many compartments they're in:

$$3 \cdot \left(\binom{8}{0} + \binom{8}{1} + \binom{8}{2} + \binom{8}{3} + \binom{8}{4} + \binom{8}{5} + \binom{8}{6} + \binom{8}{7} + \binom{8}{8} \right)$$
$$= 3 \cdot (1 + 8 + 28 + 56 + 70 + 56 + 28 + 8 + 1) = 3 \cdot 256 = 768$$


U//B/C
U//C R
U//A/C R
U//A/B/C
U//B
U R
U//A/B
U//A R
S//B/C
S//C R
S//A/B/C W
RW
S//A/C
S//B
S R
S//A/B
S//A R
TS//B/C
TS//C
W
TS//A/B/C
W
TS//A/C
TS//B
TS
TS//A
TS//A/B

2. (20 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) __N__ Situation whose result depends on which of two parallel actions happens first

   (b) __Q__ Attacker-supplied instructions implementing malicious functionality

   (c) __R__ A component whose failure could violate your security

   (d) __K__ Code that can run correctly at any memory location

   (e) __F__ Unintended means of conveying information

   (f) __M__ System call used for system call interposition

   (g) __B__ A value whose overwrite signals an attack

   (h) __D__ System call enabling a restricted file system

   (i) __E__ Secret data should not flow to public sinks

   (j) __L__ The power to take security-relevant actions

   (k) __O__ Behaving less safely because of a safety mechanism

   (l) __S__ What marketing means when they say "trusted"

   (m) __T__ Capable of simulating any program

   (n) __C__ Security token that both designates a resource and provides authority to access it

   (o) __J__ Long instruction sequence leading to shellcode

   (p) __P__ Random but not secret value added to a password hash

   (q) __A__ Lack of connection between two networks

   (r) __G__ Frame pointer

   (s) __H__ Untrusted data should not flow to critical sinks

   (t) __I__ Partial order with $\sqcup$ and $\sqcap$

   A. Air gap B. Canary C. Capability D. `chroot` E. Confidentiality F. Covert channel G. `%ebp` H. Integrity I. Lattice J. NOP sled K. PIC/PIE L. Privilege M. `ptrace` N. Race condition O. Risk compensation P. Salt Q. Shellcode R. Trusted S. Trustworthy T. Turing complete

3. (20 points) Multiple choice. Circle the letter of the correct answer.

   (a) This biometric authentication technique has low error rates when performed by a computer, but can't be checked by a person:

   A. Iris codes B. Fingernail length C. Signature matching D. Fingerprints E. Voice recognition

   This is mentioned in the discussion in Anderson.

   (b) What's "common" about the Common Criteria?

   A. Every kind of product is evaluated against the same "protection profile".
   B. Anyone can perform the certification, without special government approval.
   C. The certification applies to devices used in everyday civilian life, rather than in government or the military.
   D. A single certification is recognized by the governments of many countries.
   E. A single certification can be used for products from different vendors.

   Mentioned both in class and in Anderson.

   (c) Suppose a biometric authentication system has an equal error rate of 5%. Which of these is **not** also the case?

   A. The system will always have a lower false-positive rate than allowing users based on a random coin flip with allow probability 5%.
   B. If you configure the system to have less than 3% false negatives, the rate of false positives is at least 5%.
   C. There is no way to configure the system to simultaneously have no false negatives and no false positives.
   D. If you configure the system to have less than 5% false positives, the rate of false negatives must be 5% or more.
   E. By setting the confidence threshold appropriately, we can get false-positive and false-negative rates that are both 5%.

   E is just the definition of the equal error rate. C is true whenever the EER is non-zero, and is a general feature of biometrics. B and D both lead to contradictions with the EER definition: for instance in B, if you could have 3% false negatives and 4% false positives, the EER would be at most 4%. But for A, you can always give even a good system a high false-positive rate by setting the confidence threshold appropriately.

   (d) Which of these systems does **not** use primarily capabilities for access control?

   A. The Joe-E programming language
   B. The Caja programming language
   C. The seL4 microkernel
   D. The KeyKOS operating system
   E. The SELinux operating system

   Joe-E, Caja, and seL4 were mentioned as examples of capability systems in class. KeyKOS was mention in the paper on capabilities. SELinux by contrast was mentioned as an example of an MLS system.

(e) In the CFI paper, legal jump targets are identified by a special 32-bit value. Which special property must that value have?

A. It must be a palindrome (same big-endian as little-endian)

B. It must not contain any 0 bytes

C. It must be greater than the largest code address

D. It must not be a legal x86 instruction

E. It must not appear elsewhere in the program code

Mentioned in lecture and in the CFI paper. As long as the value is unique, any of A-D might be violated.

(f) Which one of these functions can never overflow a buffer?

A. `strcpy` B. `sprintf` **C.** `printf` D. `strlcpy` E. `strncpy`

`strcpy` *and* `sprintf` *are classic sources of buffer overflows.* `strncpy` *and* `strlcpy` *try to be safer by letting you specify the size of the buffers, but you could still have a problem if you pass the wrong size. On the other hand* `printf` *doesn't write to a user-supplied buffer at all: its results go straight to the output.* `printf` *can have format string vulnerabilities, but format string vulnerabilities aren't buffer overflows.*

(g) According to the C standard, which of these might be true of the execution of a program with undefined behavior?

A. It could appear to run normally, but with some security checks bypassed

B. It could print an error message

C. It could terminate without an error message

D. It could have a segfault

E. All of the above

Mentioned in lecture.

(h) This pair of functions, mentioned in the StackGuard paper, have a behavior like a function pointer that makes them valuable to code-injection attackers:

A. `read/write`

B. `setjmp/longjmp`

C. `chmod/chown`

D. `printf/scanf`

E. `getuid/setuid`

There's a section about this in the StackGuard paper. `longjmp` *buffers were also briefly mentioned in class as targets for control-flow injection.*

(i) In a system enforcing a W⊕X policy, which of the following areas could be executable?

A. The read-only data area

B. The BSS area (`.bss`)

C. The heap

D. The initialized data (`.data`) area

E. The stack

The most elegant way to answer this is to notice that the read-only data area is not writable, which means that it's okay for it to be executable under W⊕X. The other areas mentioned are all writable, so they can't be safely executable. In fact it's common for the read-only data section to have the same R-X permissions as the text (code) section, because the read-only data section comes right after the code.

(j) Which of these equations does **not** have a solution, if the operations are performed using 32-bit `unsigned int`s?

A. $x + 1 < x$

B. $2 \cdot x = 1$

C. $13 \cdot x = 10$

D. $2 \cdot x = 10$

E. $2 \cdot x < x$

The solutions for C and D were straight out of Exercise Set 1; also D has a solution that should be obvious. A and E are more examples of things that seem impossible that become possible because of overflow: for A take $x =$ `0xffffffff`, and for E take $x =$ `0x80000000`. On the other hand B is not possible because even with overflow, the result of multiplying by 2 is always even. If you think about $2 \cdot x$ as left shift, the new low-order bits are always zero.

4. (25 points) Non-defensive programming. The following C function was not written defensively: if it's supplied with unusual input, various bad things can happen. There are two ways of thinking about these problems: you can consider them coding mistakes in this function, or you can consider them preconditions of this function that need to be documented so that the rest of the program can use it safely.

   Find and explain three such problems with this function. Pick the three most serious problems you can find: we're looking for the sorts of problems that could be exploitable if the arguments to the function were controlled by an attacker (though you don't need to construct such an attack for this question).

   For each problem we want four pieces of information. List the line number(s) on which the problem occurs. Give a precondition, a description of the what property well-formed input should have to avoid the problem. Describe the coding mistake and consequences: what did the programmer do wrong (or at least non-defensively) and what bad outcome can occur? Finally, describe how to fix the coding mistake.

```
1  struct piece { int row; int col; char *symbol; };
2  struct piece *pieces = 0;
3
4  struct square { char *description; /* ... */ };
5  struct square board[8][8];
6
7  /* Place a bunch of pieces on the board. */
8  int load_pieces(struct piece *p, int num_pieces) {
9    int i;
10   pieces = malloc(num_pieces * sizeof(struct piece));
11   if (!pieces) {
12     fprintf(stderr, "Allocation failed!\n");
13     return 0;
14   }
15   for (i = 0; i < num_pieces; i++) {
16     char buf[20];
17     int r = p[i].row, c = p[i].col;
18     if (r >= 8 || c >= 8) {
19       fprintf(stderr, "Position out of range!\n");
20       return 0;
21     }
22     sprintf(buf, "%d x %d: %s", r, c, p[i].symbol);
23     board[r][c].description = strdup(buf);
24     pieces[i] = p[i];
25   }
26   return 1; /* Success */
27 }
```

(a) Problem 1. Line number(s): **10, 24**
Precondition:

> **Solution:** `num_pieces` should be small enough that it does not overflow when multiplied by `sizeof(strict piece)`. Given that these pieces are filling up an 8 by 8 board, a tighter precondition like `num_pieces` ≤ 64 would also be reasonable.

Coding mistake and consequences:

> **Solution:** The multiplication on line 10 could overflow without checking. If this happens, the array allocated for pieces can be smaller than the number of entries copied into it on line 24, a buffer overflow.

How to fix:

> **Solution:** Enforce a tighter precondition on `num_pieces`, or abort if an overflow occurs.

(b) Problem 2. Line number(s): **18, 23**
Precondition:

> **Solution:** The `row` and `col` fields of the supplied pieces should be non-negative.

Coding mistake and consequences:

> **Solution:** The bounds check on line 18 checks the upper bound on `r` and `c`, but fails to check the lower bound, perhaps because the programmer hadn't noticed the variables were signed. Thus any negative value for the row or column is allowed, but will cause a value outside the array to be overwritten on line 23.

How to fix:

> **Solution:** Add conditions `r < 0 || c < 0` to the `if` statement on line 18.

(c) Problem 3. Line number(s): **16, 22**
Precondition:

> **Solution:** The `symbol` field of each piece should be a short string. For instance less than 10 characters would be safe.

Coding mistake and consequences:

> **Solution:** The code calls `sprintf` with a buffer that may not be big enough to hold the generated string. If the symbol string is long (or if `r` and `c` are long negative numbers, for that matter), the buffer `buf` will overflow.

How to fix:

Solution: Probably best is to use `snprintf` instead of `sprintf`, which won't overflow its buffer. You can also use a format specifier like `%.10s` to limit how much of the symbol is included in the string, or make a copy of just the first few characters of the symbol string.

5. (20 points) Return-oriented shellcoding. As part of a larger attack, you need to use ROP to create some code to call the Linux `mprotect` system call and disable W⊕X protection on the rest of your shellcode. In particular you've figured out that the call you want to make would look like:

```
mprotect(stack_pointer - 10000, 20000, PROT_WRITE|PROT_EXEC);
```

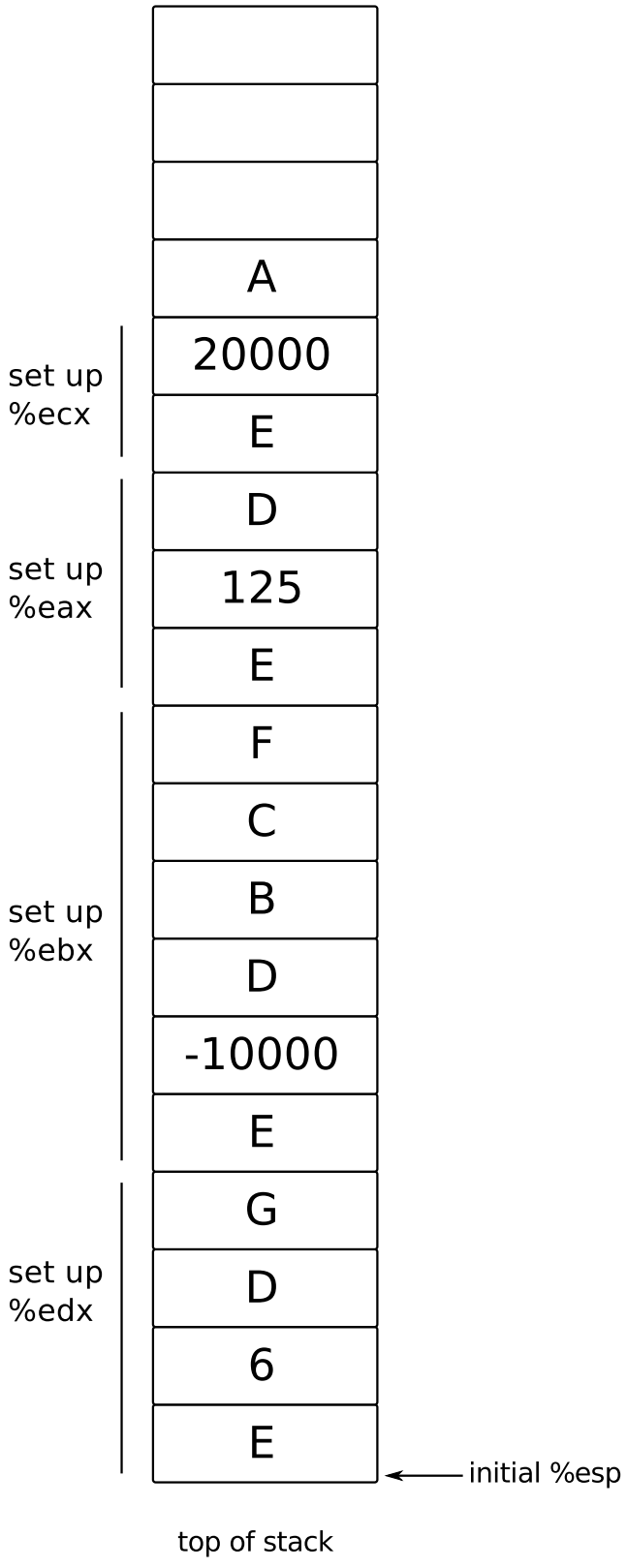
if you could write it in C. (`stack_pointer` can be the value of `%esp` at any point in the shellcode, since the margin of 10000 bytes on either side is enough to take care of any minor variation.) Looking up in the appropriate header files, you've also discovered that `mprotect` is system call number 125, and that the numeric value of the protection flags is $2|4 = 6$. According to the Linux system call calling conventions, you need to put the system call number in `%eax`, put the three arguments in the registers `%ebx`, `%ecx`, and `%edx` respectively, and then execute the instruction `int 0x80`.

You've also found a number of useful-looking gadgets, which are shown on the right side of the next page. Your job is to fill in the return-oriented program on the picture of the stack shown on the left side of the next page, starting from the "top" of the stack at the bottom of the page. Each space on the stack represents a 32-bit value. You can fill it in with a fixed number by writing the number in the box, or you can make it a pointer to a gadget by writing the letter of the gadget in the box. (When drawing these diagrams in the past we've drawn an arrow from the box to the gadget, but letters should be easier to read.) You can use each gadget as many or as few times as you would like: our solution uses all of them, some several times.

Hint: plan carefully the order in which you fill in the registers, since some operations can only be done using certain registers for intermediate values.

It's possible to achieve your goal using only the gadgets shown, and without using all the stack spaces we've drawn for you. But if you can't figure out how, you can earn partial credit with a solution in which you invent new gadgets (write them below the ones we put there with their own letters). But you'll lose one point for each instruction in a new gadget you use.

We've written the x86 instructions in AT&T syntax, so the result operands are on the end. For instance `mov x, y` is like `y = x`, and `add x, y` is like `y += x`.




```
A. int 0x80; ret
B. mov %esp, %ebx; ret
C. add %ebx, %eax; ret
D. mov %ecx, %eax; ret
E. pop %ecx; ret
F. mov %eax, %ebx; ret
G. mov %eax, %edx; ret
```