Computer Science 5271
Fall 2015
Midterm exam (solutions)
October 19th, 2015
Time Limit: 75 minutes, 4:00pm-5:15pm

- This exam contains 8 pages (including this cover page) and 4 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 devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TA, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that 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 5:15pm. Good luck!

Your name (print): ____________________

Your UMN email/X.500: ____________________`@umn.edu`

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

Sign and date: ____________________

Question	Points	Score
1	20	
2	36	
3	24	
4	20	
Total:	100	

1. (20 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) __**G**__ Windows acronym for W $\oplus$ X

   (b) __**L**__ Canary value location

   (c) __**B**__ Security token that both designates a resource and provides authority to access it

   (d) __**K**__ Stack pointer register

   (e) __**M**__ Attack technique that requires overlapping instructions

   (f) __**C**__ Windows acronym for a CFI-like defense

   (g) __**O**__ An invariant true on calling a function

   (h) __**D**__ Unix system call to change file permissions

   (i) __**A**__ Set of allowed subjects and actions for a resource

   (j) __**I**__ Holds Linux/x86 system call number

   (k) __**Q**__ Contest to find security bugs in Google Chrome

   (l) __**T**__ CPU state with privileged instructions disabled

   (m) __**F**__ Password hashing function

   (n) __**N**__ Page-table bit that denies execute permission

   (o) __**S**__ Modifying code so it can run at a new memory base address

   (p) __**P**__ The power to take security-relevant actions

   (q) __**R**__ Attack technique based on instruction gadgets

   (r) __**E**__ Unix system call to change file UID and GID

   (s) __**J**__ Amount of randomness

   (t) __**H**__ Reference to free()d memory

   A. ACL B. capability C. CFG D. `chmod` E. `chown` F. `crypt` G. DEP H. dangling pointer I. `%eax` J. entropy K. `%esp` L. `%gs:0x14` M. JIT spray N. NX O. precondition P. privilege Q. Pwnium R. ROP S. relocation T. user mode

2. (36 points) Multiple choice. Each question has only one correct answer: circle its letter.

   (a) Which of these defense techniques would completely prevent a ROP attack from returning from an intended return instruction to an unintended gadget?

      A. ASLR

      B. A non-executable stack

      C. Adjacent stack canaries

      D. A shadow stack

      E. A and C, but only if used together

   A shadow stack ensures that each (compiler-intended) return targets only the site of the corresponding call. ROP is not affected by the stack being non-executable, because it executes only bytes from the code section. ASLR and adjacent canaries would both make ROP or entering ROP code more difficult, but even together they are not a complete defense.

   (b) What two methods are mentioned in the StackGuard paper to prevent canary forgery?

      A. "terminator canary" and "random canary"

      B. "StackGhost" and "random XOR canary"

      C. "stack layout randomization" and "entropy canary"

      D. "StackGhost" and "PointGuard"

      E. "Keccak" and "Rijndael"

   StackGhost is a real name of a system, but not one we've discussed. Stack layout randomization would also make it harder to exploit buffer overflows, but it's not related to canary forgery want was not proposed in the StackGuard paper. Keccak and Rijndael are crypto primitives, now better known as SHA-3 and AES respectively.

   (c) Which of the following functions found in the C library could *not* be used as a dispatcher function in control-flow bending?

   A. `fputs` **B.** `getenv` C. `printf` D. `strcat` E. `memcpy`

   As mentioned in the control-flow bending paper, the other functions can all have their arguments chosen so that they write an attacker-controlled value to a memory location. By contrast `getenv` *simply returns a pointer to a pre-existing memory area without writing anywhere.*

   (d) Suppose we are using a MLS system with levels unclassified < classified < secret, and enforcing a high-watermark policy. If a program has read classified data, what level(s) can it write to afterwards?

      A. Classified, only

      B. Unclassified, only

      C. Classified and all levels below classified

      D. Classified and all levels above classified

   This is the "no write down" part of the policy.

   (e) If the login process requires both "something you have" and "something you know", it is using a model called:

      A. Hash-and-salt mechanism

      B. Two-factor authentication

      C. Microsoft Passport

D. Two-channel authentication

E. Biometric authentication

We never discussed anything called two-channel authentication; some researchers use that phrase to refer to challenge-response authentication using both a PC and a mobile phone. The other wrong answers are unrelated concepts in authentication.

(f) To prevent a directory traversal attack, ensure that untrusted file paths do not contain:

A. `/etc/passwd` B. `././/` C. `/////etc` D. `/etc/sudoers.d` **E.** `../`

Any of these strings might appear in an attack, but the essence of directory traversal is escaping from an intended directory, so `..` *is the only component that would be effective to blacklist.*

(g) Which of these could be a reason why control-flow integrity has been slow to be widely deployed?

A. The original CFI technique did not allow for CFI-protected code to call non-CFI-protected libraries.

B. The original CFI techniques only worked for RISC architectures.

C. CFI was first invented at Microsoft, so it doesn't work on Unix.

D. CFI requires a shadow stack, which is incompatible with C++ virtual methods.

E. Even when implemented in an optimizing compiler, CFI more than doubles memory usage.

The first SFI techniques worked only for RISC, but the initial CFI paper was for x86. Some of the inventors worked for Microsoft, but many later systems have provided CFI protection on Unix. CFI can be used together with a shadow stack, but doesn't require it, and there is no incompatibility between a shadow stack and virtual methods. CFI checks can increase the size of a program's code, but usually by much less than a factor of two, especially when implemented in a compiler.

(h) The database that tells which students are in which group for hands-on assignment 1 is stored in a file on the CSE Labs machines:

```
% ls -l groups.db
????????? 1 nishad student 727 Sep 29 15:51 groups.db
```

The file should be able to be edited by the TA Nishad, scripts running as students (in group `student`) should be able to do lookups, and the professor (not in group `student`) should be able to check it. But students should not be allowed to change the database themselves. Which of these would be an appropriate permissions mode for the file (replacing the question marks)?

A. `04755, -rwsr-xr-x`

B. `00644, -rw-r--r--`

C. `00664, -rw-rw-r--`

D. `00777, -rwxrwxrwx`

E. `00640, -rw-r-----`

Other-read should be enabled for the professor, group-write should be disabled because of the students, and no execution or set-uid bits make sense for a database.

(i) A lattice is a good mathematical model for permissions in a multi-level secure system with compartments because it has all of the following properties *except*:

A. Reflexive, so that everyone can share information with themselves
B. Least-upper bound, to compute the permissions when data values are combined
C. Total order, so that every pair of subjects can communicate in one direction or the other
D. Transitive, so that information that can flow via a third party can also flow directly
E. Antisymmetric, since if two subjects can share information in both directions they can be treated as equivalent

It is because lattices are not totally ordered that are a good model in the presence of multiple compartments.

(j) In an x86 format-string attack, the address that a `%n` specifier will write to needs to be stored in the:

A. data section B. kernel C. heap D. text section **E. stack**

The function's arguments are on the stack, so `%n` can only overwrite stack locations.

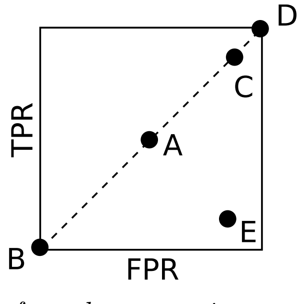
(k) The Joe-E language builds an object-capability access control system on top of the object system of Java. C++'s object system is similar to Java's in some ways, but a similar design based on C++ would be insecure because of this C/C++ feature:

A. `unsigned` integers
B. `int`-to-pointer casts
C. `goto`
D. pointer-to-`int` casts
E. `setjmp`/`longjmp`

The ability to cast integers into pointers allows a C/C++ program to forge a pointer to an arbitrary object if the program can guess its address. The other features are also missing in Java, but don't present a similar danger.

(l) Five research projects in biometric authentication produced systems with the following empirical results. Which system is worthy of future research?

A. 50% true positive rate, 50% false positive rate
B. 0% false positive rate, 100% false negative rate
C. 90% true positive rate, 10% true negative rate
D. 100% true positive rate, 100% false positive rate
E. 5% true positive rate, 95% false positive rate



Systems A-D all lie on the line of random guessing, so they could just be reimplemented with nothing more than a random number generator. E's performance is even worse, far below that line, but this means that if you just swapped its accept and reject answers you'd get a reasonable system: that would be worth more investigation.

3. (24 points) Avoiding buffer overflows with invariants. Below are two implementations that use loops to convert a string into a version with non-printable characters replaced by backslash escapes like `\x7f`. In each one, we've left out parts of the code, and/or invariant properties related to avoiding overflowing the output buffer. Fill in the blanks so the code works correctly. The invariants should be mathematical properties that are true, and explain why the buffer accesses to `out` can't overflow (you may use either C-style or math-style notation). For some blanks we've given suggestions about what variables to use.

   The functions all have the same specification: to translate a null-terminated string `s` (0 to 10000 bytes long) into a version that uses C backslash escapes. The return value is dynamically allocated and the caller should `free` it. The functions return a null pointer if allocation fails. The definition of the `escape_char` function, which escapes a single character, is on the next page.

   (a) Approach 1: using `realloc`. Recall that `realloc` is a function that changes the size of a `malloc`-allocated buffer.

```
char *escape1(const char *s) {
    int i;
    int j = 0;
    size_t out_size = strlen(s) + 1;
    char *out = malloc(out_size);
    if (!out)
        return 0;
    for (i = 0; s[i] != '\0'; i++) {


        if (    j > out_size - 5     ) {
            out_size += 5;
            out = realloc(out, out_size);
            if (!out)
                return 0;
        }
        /* Invariant property: out_size is the size of the buffer
           pointed to by "out" */


        /* Invariant:        j <= out_size - 5
                     (relating j and out_size) */
        escape_char(s[i], out, &j);
    }
    out[j++] = '\0';
    return out;
}
```

(b) Approach 2: conservative allocation.

```
char *escape2(const char *s) {
    int i;
    int j;
    char *out;


    out = malloc(   4 * strlen(s) + 1     );
                 /* (use strlen(s)) */
    j = 0;
    if (!out)
        return 0;
    for (i = 0; s[i] != '\0'; i++) {



        /* Invariant:     j <= 4 * i
                         (relating i and j)  */
        escape_char(s[i], out, &j);
    }
    out[j++] = '\0';
    return out;
}
```

The following helper functions are used by both implementations:

```
/* Convert an int 0 <= i <= 15 into an ASCII hex digit: 0-9 or a-f */
char hex_digit(int i) {
    if (i < 10) { return '0' + i; }
    else { return 'a' + (i - 10); }
}

/* Write an escaped version of character C into the buffer OUT, using
   and updating index *J. Increments *j by between 1 and 4. */
void escape_char(unsigned char c, char *out, int *j) {
    if (c == '\n') {
        out[(*j)++] = '\\'; out[(*j)++] = 'n';
    } else if (c == '\\') {
        out[(*j)++] = '\\'; out[(*j)++] = '\\';
    } else if (c >= ' ' && c <= '~') {
        out[(*j)++] = c;
    } else {
        out[(*j)++] = '\\'; out[(*j)++] = 'x';
        out[(*j)++] = hex_digit(c >> 4);
        out[(*j)++] = hex_digit(c & 0xf);
    }
}
```

4. (20 points) Attack, defense, and counter-attack. Low-level software security has seen a lot mutual evolution of attacker and defender techniques. This question traces this back-and-forth influence. Here are some examples of attacker and defender techniques:

Attacker techniques	Defender techniques
A. shellcode in environment variable	
B. return address overwrite	J. W $\oplus$ X
C. directory traversal	K. stack canary
D. pointer disclosure	L. shadow stack
E. return-oriented programming	M. ASLR
F. heap spray	N. path sanitization
G. non-control data overwrite	O. control-flow integrity
H. return to libc	
I. call-preceded ROP	

Using the pairs of blanks below, in the left column give five examples of an attack leading to defense that block the attack. Then in the right column give five examples of a defense leading to a counter-attacks that circumvents the defense. Fill in the blanks with the letters of techniques in the table above. Order is important: the technique on the left of the arrow is the one that came first, and the one on the right is the reaction from the opposing side. More than ten such linkages are possible, so choose ones where the relationship is clearest and most direct, and you don't have to use all the techniques. But you will want to use some techniques more than once.

Attack (A-I) $\rightarrow$ defense (J-O)	Defense (J-O) $\rightarrow$ counter-attack (A-I)
__**A**__ $\rightarrow$ __**J**__	__**J**__ $\rightarrow$ __**E**__
__**A**__ $\rightarrow$ __**M**__	__**M**__ $\rightarrow$ __**F**__
__**A**__ $\rightarrow$ __**O**__	__**M**__ $\rightarrow$ __**D**__
__**C**__ $\rightarrow$ __**N**__	__**O**__ $\rightarrow$ __**G**__
__**H**__ $\rightarrow$ __**O**__	__**O**__ $\rightarrow$ __**I**__

Here's a non-exhaustive list of other answers that were also worth full credit. Attack to defense: $B \rightarrow K$, $B \rightarrow L$, $E \rightarrow O$, $E \rightarrow L$, $H \rightarrow M$. Defense to counter-attack: $J \rightarrow H$, $K \rightarrow G$, $L \rightarrow G$.