Computer Science 5271
Fall 2021
Midterm exam (solutions)
October 25th, 2021
Time Limit: 75 minutes, 2:30pm-3:45pm

- 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.
- This exam contains 10 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing, and write your username in the upper-right corner of each subsequent 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 TAs, 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. But write only on the printed sides of exam pages.
- 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 3:45pm. 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	30	
2	20	
3	30	
4	20	
Total:	100	

1. (30 points) Multiple choice. Each question has only one correct answer: circle its letter.

   (a) Suppose that the UMN IT managers are considering increasing the minimum password length from 16 to 24 characters, to improve resistance to offline guessing attacks. Suppose a user has selected a 24-character password in which each character is any uppercase letter, lowercase letter, or digit independently and with equal probability (e.g., `SQMq1bvXfddhhGH24W2DV2fv`). Which of these formulas gives the total number of possible passwords?

   A. $26^{24} + 26^{24} + 10^{24} \approx 2^{114}$
   B. $24^{26} + 24^{26} + 24^{10} \approx 2^{120}$
   C. $(26 + 26 + 10)^{24} \approx 2^{143}$
   D. $24^{(26+26+10)} \approx 2^{284}$
   E. $26^{24} \cdot 26^{24} \cdot 10^{24} \approx 2^{305}$

   There are 62 choices for each character, and the choices for each character are independent, so they multiply across all the positions, which is the same as exponentiation. If you are not feeling confident in remembering in which direction the exponentiation goes, you might try thinking through a smaller familiar example, like that if you have three characters that are each digits, there are 1000 possibilities.

   (b) An attacker with access to a million devices, each of which can check a billion passwords per second, can try about 2^{75} passwords in a year. Whichever of the computations in the previous question is correct, this suggests that an attack against a password chosen according to the process in that question would be infeasible. However, many objections might still be raised to a policy of requiring 24-character passwords containing letters and digits. Which of the following is **not** a reasonable criticism?

   A. Users will make more typos with 24-character passwords.
   B. Users will have a hard time remembering 24-character passwords.
   C. A password cannot be secure unless it contains punctuation characters like `(` or `@`.
   D. Users will not generate their passwords uniformly at random.
   E. A 16-character length would be enough to resist guessing attacks.

   If you repeated the calculation from the previous question with an exponent of 16 instead of 24, the exponent would be two-thirds, namely around 2^{95}, which is still comfortably larger than 2^{75} which came from some generous assumptions. A, B, and D all raise valid usability concerns about long passwords. But C is not sensible. Asking users to choose passwords from a larger set of characters can increase security if they are used randomly, since it increases the base in a computation like the one in the previous question. But making passwords longer also increases entropy; if a password is long enough, it can have high entropy even with a more limited character set.

   (c) The following properties are true for all values of 32-bit two's complement integer values `x` and `y` (i.e., C `int`s), **except**:

   A. `x + 1 != x`
   B. `5*x == (x << 2) + x`
   C. `x * y == y * x`
   D. `x + -x == 0`
   E. `-x <= x`

E fails if `x` *is already negative and* `-x` *is positive. For A, notice that even if there is overflow, the low bits of* `x + 1` *and* `x` *are always different. For B, left shifting by 2 is the same as multiplying by 4.*

(d) Consider the set of positive integers where the relationship $a \sqsubseteq b$ is defined to hold when a divides b evenly (i.e., there is another positive integer c such that $ac = b$). This operation forms a partial order. If we want to make it a lattice, what operation should be the meet $a \sqcap b$?

A. $\min(a, b)$
B. $\max(a, b)$
C. $\lfloor (a + b)/2 \rfloor$ ($\lfloor x \rfloor$ represents rounding down to the nearest integer)
D. $\gcd(a, b)$
E. $a + b$

The meet is also called the greatest lower bound. $a \sqcap b$ should be less than a and less than b, but greater than or equal to any other value that is also less than both of them. In a diagram, it's the closest point below a and b where they come together. In the case of divisibility, it needs to be a divisor of a and a divisor of b (a common divisor), but it should be the largest among all the common divisors, which is just the definition of the GCD (greatest common divisor). You might also recall that if you represent a and b in their prime factorization, you get the GCD by taking only the divisors that they have in common.

(e) This number x is the multiplicative inverse of 3 mod 2^{32}, i.e. $3 \cdot x \equiv 1 \pmod{2^{32}}$. (Possibly relevant facts: `3*x = x + (x << 1)`; `0xffffffff = 3*0x55555555`; `3*6667 = 20001`)

A. `0x33333333` B. `0x55555555` C. `0x80000003` **D.** `0xaaaaaaab` E. `0xfffffffd`

The most direct way to choose among these answers is to just the compute just the lowest hex digit of multiplying each of them by 3, which is just computation mod 16. $3 \cdot 3 = 9 \pmod{16}$, $5 \cdot 3 = 15 \pmod{16}$, $11 \cdot 3 = 33 = 1 \pmod{16}$, *and* $14 \cdot 3 = 42 = 10 \pmod{16}$. *The low hex digit has to be 1 if the entire product is going to be* `0x00000001`. *If you want to think about the multiplication in binary, the first fact reminds you that it is equivalent to shifting and adding. The second fact is helpful if you want to solve the equation* $3 \cdot x = 2^{32} + 1$ *because it tells you that* $2^{32} = 1 \pmod{3}$, *so* $2 \cdot 2^{32} + 2$ *will be a multiple of 3. The third fact is the solution to an analogous version of the problem in decimal (mod* 10^4*): 6667 is two thirds of 10,000, rounding up, while answer D is two thirds of* 2^{32}*, rounding up.*

(f) Compared to x86-32, x86-64 has several features that make things easier for defenders and harder for attackers. Which of these is **not** such a difference?

A. The stack grows upwards, so a buffer overflow can't overwrite the return address
B. Expanded RIP-relative addressing and more registers allow more efficient PIC
C. Arguments in registers cannot be corrupted via a stack buffer overflow
D. A larger address space allows ASLR to have more entropy
E. All valid x86-64 user-space addresses contain null bytes

B-E are all true features of current x86-64, or its standard calling convention in the case of C. E holds because only the low 48 bits of addresses may be non-zero. The premise of A is false: the x86-64 stack grows downwards the same as the x86-32 stack. Changing

the direction of the stack, while keeping arrays growing towards higher addresses as well, would make it so that overflowing an array in the positive direction could not overwrite the return address of the same function, though the stack frames of more recent functions might still be corrupted.

(g) When logging on to the course Canvas page, you must use a registered smartphone or security token in addition to providing a password. This is an example of:

A. Multi-factor authentication
B. Separation of duty
C. A CAPTCHA
D. Single sign-on
E. Biometric authentication

You might have noticed that the name of the product that enforces this feature, "Duo", is based on "two-factor authentication"; any number of factors more than 1 is "multi-factor."

(h) Suppose you want to use a format-string vulnerability to cause a program to output a lot of output, say n characters, with a short format string. About how long of a format string is required?

A. 2 characters
B. $\log_2(\log_2 n)$ characters
C. $\log_{10} n$ characters
D. $\sqrt{n}$ characters
E. $(4/16) \cdot n$ characters

This question is intended to capture a technique that you should have encountered in Exercise Set 1. A more verbose way of describing the question would have been to say that we are interested in a general technique which can produce an output of any desired length, and among all such techniques we're interested in the one that needs the shortest format string. For A, you might be thinking of `%s`*. If you are lucky that the appropriate argument location holds a pointer to a large memory region without nulls,* `%s` *can cause many characters to be printed, but if you're unlucky you might get only a few characters or the program might just crash. The formula in E is achievable for instance by using* `%0lx`*, which is 4 characters long and prints 16 hex digits on a system like x86-64 where* `long` *is a 64-bit type. But C is also achievable by specifying padding on a numeric format specifier, as in* `%999d`*, and requires a much shorter format string. I don't know of any format string patterns that would match the formulas in B or D. (Note that the formula choices were listed in increasing order.)*

(i) On the CSE Labs machines, the standard umask is `0077`, which causes files to be created without permissions for other users. This is an example of which of the following secure design principles?

A. Complete mediation
B. Open design
C. Fail-safe defaults
D. Least privilege
E. Economy of mechanism

You might have noticed we were careful to avoid using the word "default" in the question, but it would have been doubly appropriate: the umask represents the default permissions for

files that programs can further restrict, and `0077` *is the default value of the umask. You can change the permissions on your files to anything you want, but the system administrators decided it would be best if files start out private, and are made public only if you explicitly choose for them to be. "Least privilege" might also sound plausible, because this choices causes other users to have fewer privileges to access your files than if a different umask were used. But the principle of least privilege is about selecting this privileges that a subject should have, based on what is needed for its normal operation. The umask is a setting related to objects rather than subjects, and it's a default for when you haven't given thought to what the appropriate setting should be.*

(j) For an attack to be possible, a use-after-free bug usually needs to involve a memory region that was first allocated with one type ("type 1") being reused with a different type ("type 2"). Suppose there is an attack possible where a value is written with type 1 before the value is reallocated, then read with type 2 later. For this attack to work, what other problem must the program have?

A. Null pointer dereference
B. Format string vulnerability
C. Race condition
D. Infinite recursion
E. Reading uninitialized data

C doesn't make any guarantees about the contents of memory allocated with `malloc`*: it might contain any values, in particular because it might have been reused from another purpose. Because of this feature, correct programs need to do their own initialization of memory from* `malloc`*, and this initialization will overwrite the reused contents. The ordering of events mentioned in the question, where a write happens before reallocation and a read happens after, would only be useful to the attacker if the program also didn't follow this initialization rule.*

2. (20 points) Matching vulnerability types.

   On the left side are ten examples of C code patterns that represent various security-relevant bugs. Fill in the blank next to each with a letter corresponding to the name of a vulnerability type from the right side. Every vulnerability type is used exactly once, except that "Buffer Overflow" is used exactly two times. You can assume any variables named `evil` or `bad` might be under the control of an attacker, and have not had relevant security checks performed. Ellipses `...` represent omitted code.

   (a) __**C**__
   ```
   num_objs = evil;
   p = malloc(num_objs * sizeof(obj));
   ```

   (b) __**A**__
   ```
   p = q = malloc(...);
   free(p);
   q->field = evil;
   ```

   (c) __**I**__
   ```
   char buf[30];
   for (i = 0; i < bad; i++)
       buf[i] = evil[i];
   ```

   (d) __**H**__ `system("cp a b");`

   (e) __**G**__
   ```
   free(p);
   ...
   free(p);
   ```

   (f) __**D**__ `printf(evil, 42);`

   (g) __**I**__ `gets(buf);`

   (h) __**B**__
   ```
   if (access(fname, R_OK) == 0)
       fd = open(fname, O_RDONLY);
   ```

   (i) __**F**__
   ```
   char path[] = "/tmp/x.XXXXXXX";
   mktemp(path);
   fd = open(path, O_WRONLY|O_CREAT);
   ```

   (j) __**E**__
   ```
   snprintf(buf, sizeof(buf),
            "/dir/%s", evil);
   fd = open(buf, O_RDWR);
   ```

   A. Use after free
   B. TOCTTOU race
   C. Integer overflow
   D. Format string vulnerability
   E. Directory traversal
   F. File creation race
   G. Double free
   H. Insecure PATH dependency
   I. Buffer overflow
   I. Buffer overflow

3. (30 points) Buffer overflow attack.

   The following function from a Linux/x86-64 program has a buffer overflow vulnerability. In this question, you'll figure out the stack layout of the function and what data should be used in the overflow to build a successful attack.

   Specifically, assume that normally the function would return to the address `0x4011fb`, and that the argument `s` points to a string under the attacker's control. Your goal as the attacker it to make the execution instead jump to the address `0x4d5271`, where you have arranged for some shellcode to exist. Below are excerpts of the relevant code in C and assembly language.

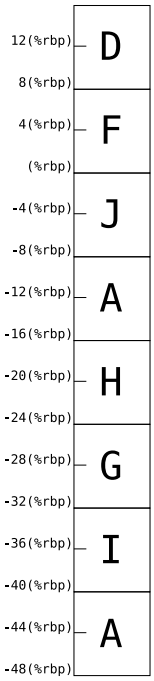
```
int global_fd = -1;

void func(char *s) {
    int fd = 0;
    char buf[16];
    fd = open("/tmp/foo", O_RDONLY);
    strcpy(buf, s);
    if (fd != global_fd) {
        exit(1);
    }
}
```

```
.LC0: .string "/tmp/foo"
func: pushq   %rbp
      movq    %rsp, %rbp
      subq    $48, %rsp
      movq    %rdi, -40(%rbp)
      movl    $0, -4(%rbp)
      movl    $0, %esi
      movl    $.LC0, %edi
      movl    $0, %eax
      call    open
      movl    %eax, -4(%rbp)
      movq    -40(%rbp), %rdx
      leaq    -32(%rbp), %rax
      movq    %rdx, %rsi
  ==> movq    %rax, %rdi
      call    strcpy
      movl    global_fd(%rip), %eax
      cmpl    %eax, -4(%rbp)
      je      .L3
      movl    $1, %edi
      call    exit
.L3:  nop
      leave
      ret
```

   Use this code to answer the questions on the following page.

   Recall that `strcpy` copies a sequence of characters pointer to by its second argument to the location pointed to by its first argument, up to a null terminator. The `open` system call opens the file specified in its first arguments for the operation(s) specified in its second argument, returning a non-negative file descriptor if successful or -1 on an error. Assume that the variable `global_fd` still has the value -1 when the attack occurs.

   (a) First, let's draw a diagram of the function's stack frame layout. In particular, draw the layout right before the call to `strcpy`, at the location marked with an arrow in the assembly code. On the left is a blank picture of a stack frame, broken into 8-byte segments and labelled by offsets relative to the frame pointer `%rbp`. On the right is a list of descriptions of contents that might appear in each segment. Fill in each box on the left with a letter from the right. Some descriptions might be used more than once, others not at all.



A. Unused/padding

B. `global_fd` and unused/padding

C. Stack canary

D. Return address

E. Saved `%rbx`

F. Saved `%rbp`

G. `buf[0 .. 7]`

H. `buf[8 .. 15]`

I. `s`

J. `fd` and unused/padding

The oldest part of the stack frame, at the top in this figure, is set up when the function starts execution. The first two instructions push a copy of the calling function's `%rbp` *on the stack, and then copy the stack pointer pointing at that location back to* `%rbp`*, which sets up that* `%rbp` *(offset 0) points at the saved* `%rbp` *F, and that is the reference point for the rest of the numbering. The only thing written to the stack earlier than that is the return address that was written by the calling function, which is above it (D). The other stack accesses go via* `%rbp`*. The variable* `fp` *is copied from* `%rax` *as the return value of* `open` *and accessed again in the* `if` *statement comparison: this matches the usage of* `-4(%rbp)`*. -4 is not a multiple of 8, because an* `int` *is only 4 bytes while the stack slots are 8 bytes, so* `fd` *is stored in the top half of the box and the lower half is unused, J. The variable* `s` *is copied from the (first and only) function parameter in* `%rdi`*, and then copied to be the second argument (*`%rsi`*) to* `strcpy`*. That matches the usage of* `-40(%rbp)`*, so I goes there.* `buf` *is 16 bytes long, so it will extend across at least two boxes, and addresses are increasing upwards in this figure, so G should be below H. In particular* `buf` *is passed as the first argument to* `strcpy`*, and you can see that this is computed as* `-32(%rbp)` *(the code uses* `lea` *instead of* `mov` *because what is being passed is a pointer to the memory region, not its contents). Thus G goes in the box starting at* `-32(%rbp)` *and H in the box above it. The remaining locations in the stack frame are not accessed, so they get A.* `global_fd` *does not appear in the picture because, as the name suggests, it is a global variable stored in the data segment rather than on the stack. This code was not compiled with stack canaries, which will make the next part easier. The caller's* `%rbx` *does not need to be saved because this function doesn't need to use* `%rbx` *itself.*

(b) Now, show what contents for the string `s` should be used to create a successful attack that hijacks control flow. Each blank below represents one character in the attack string, in order of increasing address. Fill in each blank with one character. You can write printable ASCII characters like letters as themselves, and for non-printable characters use C escapes like `\0` for null, `\n` for newline, or `\xfa` for a byte with hex value `0xfa`. You may not need to use every blank. The first 16 characters of the string that will go inside the bounds of `buf` won't be part of the attack, so we've filled them in with regular letters.

A	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B
x	x	x	x	x	x	x	x	x	x	x	x	\xff	\xff	\xff	\xff
x	x	x	x	x	x	x	\x71	\x52	\x4d						

Each group of 8 blanks here should line up with one of the blocks in the previous diagram, where the blocks pre-filled with A and B correspond to G and H respectively. In our sample answer we've used lowercase `x` *in places where any non-null byte value could be used: this includes the unused space A, the unused half of the block J that also contains* `fd`*, and the saved* `%rbp`*. If the attack succeeds it doesn't matter if the saved* `%rbp` *is corrupted, because it will never be used. In order to prevent the* `if` *comparison from exiting the program,* `fd` *should be overwritten with a value that makes it always equal to* `global_fd` *or -1. That is achieved by making each byte* `0xff` *in hex, which is the all 1-bits value. It might seem like a problem that the full value we want to overwrite the return address with, which you could write as* `0x00000000004d5271` *to emphasize that it is 64-bits, contains null bytes that will cause* `strcpy` *to stop copying. However, this turns out not to be a problem because the top bytes of the old and new values are the same and don't need to be changed. It's enough to overwrite the three bytes that need to be changed, least-significant first because x86-64 is little-endian, and then the terminating null byte can overwrite the next byte that should be 0 anyway. You could have also written the last byte as* `\x00` *for emphasis, but note that it wouldn't have worked to have a null byte anywhere in the middle of the attack string, because the bytes after it wouldn't be copied.*

Note that because we graded the two halves of this question separately in Gradescope, we graded your answers for part (b) just against what attacks would work, not for consistency or inconsistency with your answer in part (a).

4. (20 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

(a) __**R**__ A value that would cause copying to stop

(b) __**S**__ A bug allowing memory reuse with a different type

(c) __**A**__ Not connecting networks at different levels

(d) __**E**__ An attack that can be in binary or interpreted software

(e) __**J**__ An instruction that has no side-effects

(f) __**K**__ A memory permissions bit used to implement W $\oplus$ X

(g) __**Q**__ A program that runs with the identity of its file owner

(h) __**T**__ The group of users who can become the superuser

(i) __**L**__ Slightly modifying an OS kernel to run better in a VM

(j) __**D**__ An abstract model for MLS confidentiality

(k) __**B**__ Privilege based on identity, not a capability

(l) __**I**__ A measure of the uncertainty in a probability distribution

(m) __**H**__ An attack accessing unintended parts of the filesystem

(n) __**P**__ The standard username for UID 0

(o) __**F**__ Binary-only software without symbol information

(p) __**M**__ For example, $\subseteq$ on sets

(q) __**O**__ A mail server designed for increased security

(r) __**C**__ A defense that changes the base address of a memory region

(s) __**N**__ An invariant true about the results of a function

(t) __**G**__ Unintended communication with cooperating sender and receiver

A. air gap B. ambient authority C. ASLR D. Bell-LaPadula E. code injection F. COTS G. covert channel H. directory traversal I. entropy J. NOP K. NX L. paravirtualization M. partial order N. postcondition O. Postfix P. `root` Q. setuid R. terminator canary S. use after free T. `wheel`