Computer Science 5271
Fall 2024
Midterm exam
October 23rd, 2024
Time Limit: 75 minutes, 1:00pm-2:15pm

- 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. Don't put any of your answers on this page.
- 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.
- 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.
- 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!

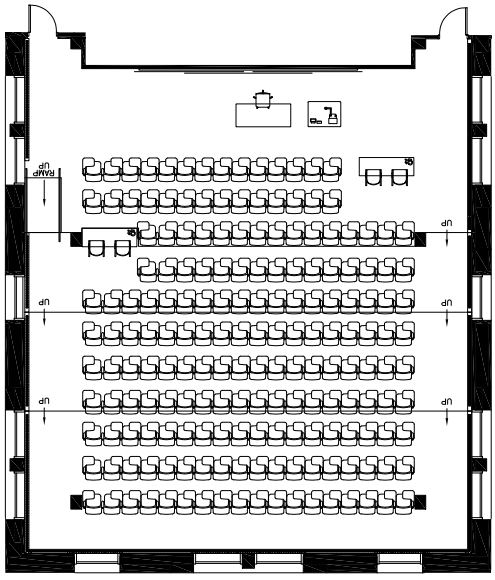
Your name (print): _____

Your UMN email/X.500: _____@umn.edu

Sign and date: _____

Question	Points	Score
1	30	
2	20	
3	30	
4	20	
Total:	100	

Mark the icon corresponding to your seat:



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

   (a) The number 20 in decimal is represented as `0x14` in hex. All of the following pairs of 32-bit integers, shown in hex, would produce a value of `0x14` when multiplied using the rules of C for `unsigned int` on x86-64, **except**:

   A. `0xfffffffb` · `0xfffffffc`
   B. `0x80000002` · `0x8000000a`
   C. `0x33333333` · `0x00000003`
   D. `0x00000005` · `0x00000004`
   E. `0x80000005` · `0x00000004`

   (b) Suppose a local variable whose type is an array of 100 characters contains sensitive information in the form of short (3-6 character) printable strings separated by null bytes, and assume the platform is Linux/x86-64. If the function containing the variable also has a call to `printf` that is vulnerable to a format string attack, which of these format specifiers would be the best choice for an attacker to use repeatedly to dump the entire contents of the array?

   A. `%lx` B. `%c` C. `%ho` D. `%s` E. `%x`

   (c) The set of all subsets of the letters A though J forms a lattice when the ordering operation $\sqsubseteq$ is defined to be the subset operation $\subseteq$. Under this definition, what is the least upper bound $\{A, B\} \sqcup \{E, F\}$?

   A. $\varnothing$ (the empty set)
   B. $\{A, B\}$
   C. $\{A, B, E, F\}$
   D. $\{A, B, C, D, E, F\}$
   E. $\{A, B, C, D, E, F, G, H, I, J\}$

   (d) Why would password capabilities be an appropriate kind of capability-based access control to be used in a large-scale networked system?

   A. Object-aggregated authority management scales to having many objects
   B. A centralized server can immediately revoke password capabilities
   C. Capabilities automatically provide ambient authority
   D. Password capabilities don't need to be managed by an OS kernel
   E. Password capabilities cannot be transferred via network messages

   (e) Random stack canaries and ASLR share all of the following features **except**:

   A. They are more resistant to guessing if they are re-randomized frequently
   B. They make some control-flow hijacking attacks more difficult
   C. They require kernel support to implement
   D. They are more resistant to guessing if they have more entropy
   E. Their protection is compromised if information leaks to an attacker

(f) All of these attack techniques were predecessors of ROP, **except**:

A. Control-flow bending
B. Return to libc
C. ret2pop
D. Chained return to libc
E. Stack smashing

(g) Suppose a program on an x86-32 platform has a hard-to-control memory safety vulnerability that leads to a return address being overwritten by a uniformly random 32-bit value. An attacker is able to set up a heap spray by allocating 1000 memory objects, each of which is 1 MiB ($2^{20} = 1048576$ bytes) long, containing a NOP sled and a 100 byte-long shellcode. These objects are placed at non-overlapping locations in the address space. If the attacker repeats the attack 10 times, what is the probability of succeeding at least once?

A. $(1 - (1000 \cdot (2^{20} - 100)/2^{32}))^{10} \approx 6\%$
B. $10 \cdot 100 \cdot 2^{20}/2^{32} \approx 24\%$
C. $(10 \cdot 100 \cdot 1000/2^{20})^{10} \approx 62\%$
D. $1 - (1 - ((1000 - 100) \cdot (2^{20})/2^{32}))^{10} \approx 92\%$
E. $1 - (1 - (1000 \cdot (2^{20} - 100 + 1)/2^{32}))^{10} \approx 94\%$

(h) All of the following situations are specified to constitute undefined behavior in the C language standard **except**:

A. Dereferencing a null pointer
B. Accessing a memory region after it has been `free()`d
C. An unhandled case in a `switch` statement
D. Accessing outside the bounds of an array
E. Integer overflow of a signed integer

(i) Applying the metric of net risk reduction implies that a security protection becomes more worthwhile when any of these happen, **except**:

A. The expected damage caused by an attack increases
B. The attack becomes more frequent
C. The cost of carrying out the attack goes up
D. The defense becomes less expensive

(j) Some laptops and smartphones now encourage users to log in via facial recognition or a fingerprint instead of with a password or PIN. These are examples of:

A. Single sign-on
B. Two-factor authentication
C. Biometric authentication
D. Compromise recording
E. CAPTCHAs

2. (20 points) A race condition attack.

   The following high-level C code attempts to copy the contents of one temporary file belonging to the Alice (username `alice`) into a new file that will also be owned by Alice. However, you may be able to see that it has TOCTTOU/race condition problems.

```
char data[32000]; size_t data_len;
void copy_alice_file(char *input_file, char *output_file) {
      /*** point A ***/
    if (!file_exists(input_file))
        print_error_and_exit();
      /*** point B ***/
    if (!is_alice_owned_and_readable(input_file))
        print_error_and_exit();
      /*** point C ****/
    if (!file_exists(output_file))
        create_alice_file(output_file);
      /*** point D ****/
    FILE *input_fh = fopen(input_file, "r");
    if (!input_fh)
        print_error_and_exit();
    read_data(input_fh, data, &data_len, sizeof(data));
    fclose(input_fh);
      /*** point E ***/
    FILE *output_fh = fopen(output_file, "w");
    if (!output_fh)
        print_error_and_exit();
    write_data(output_fh, data, data_len);
    fclose(output_fh);
}

int main(int argc, char **argv) {
    /* ... */
    copy_alice_file("/tmp/alice.in", "/tmp/alice.out");
    return 0;
}
```

   Suppose that the program containing this code runs with superuser privileges, and your goal as an attacker with the username `bob` is to trick the program into doing something else. Specifically Bob wants the program to copy the contents of the secret file `/etc/shadow`, which contains information about other users' passwords, into a file that he (Bob) can read. (To start, `/etc/shadow` is only readable by the superuser.) Assume that Bob triggers the execution of this program at a time when initially neither the input file `/tmp/alice.in` nor the output file `/tmp/alice.out` exists yet. But a different file named `/tmp/alice-recipes` owned by Alice with 0600 permissions does exist. Bob is able to run other programs, using his write access to `/tmp`, at the same time this code is running. In particular, to achieve his attack, Bob will try to get certain file system operations to occur in between the vulnerable program's operations, namely at the points marked point A through point E.

In the parts below, describe which racing attacker actions Bob should take at each point for a successful attack. You may not need to use all of the points. Suggestion: use symbolic links.

(a) At point A:

(b) At point B:

(c) At point C:

(d) At point D:

(e) At point E:

Assume that all of the functions whose names contain underscores do what their name sounds like they do: The function `file_exists` returns true if a file exists with a given pathname, and false otherwise. The function `print_error_and_exit` prints an error message and then causes the program to exit. The function `is_alice_owned_and_readable` returns true if its pathname argument is owned by `alice` and has read permissions for `alice`. Otherwise it returns false. The function `create_alice_file` creates a file with the given name suitable for storing information private to Alice: the owner of the file is `alice` and only `alice` has read or write permissions. The function `read_data` reads the contents from an open file handle into a memory buffer, keeping track of the amount of data it reads. You don't have to worry about the possibility of the file contents being bigger than the buffer. The function `write_data` is the matching operation to `read_data` and similar to the standard library function `fwrite`, writing data from memory back into a file handle open for writing.

The standard library function `fopen` opens a file handle used to read from or write to a file (specified by the second argument). It returns a null pointer if the file cannot be opened, such as if it does not exist. The standard library function `fclose` is used to close a file handle opened by `fopen`. We have left off error handling for `fclose` because it is not important to this vulnerability.

3. (30 points) Function preconditions.

   Each of the following short C functions performs some operations that are potentially unsafe, but could be performed correctly if appropriate properties of the function arguments, *preconditions*, are checked by the code calling the function. For each function, write one or more preconditions that are sufficient to guarantee that no safety or security problems will happen when running the function, but allow appropriate uses of the function to occur. Use the syntax of C to represent the preconditions whenever possible, and in other cases write clear text such as might appear in documentation.

   You don't have to mention any properties that are already checked with the `assert` function inside the function, and the number of blank lines is not intended to signal the number of preconditions we expect. We've done the first function as an example.

   `n >= 0`

```
unsigned int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

   (a) _____

```
void strcat3(char *buf, const char *s1, const char *s2, const char *s3) {
    assert(buf && s1 && s2 && s3);
    strcpy(buf, s1);
    strcat(buf, s2);
    strcat(buf, s3);
}
```

   (b) _____

```
long *alloc_and_zero_array(size_t size) {
    long *p = malloc(size * sizeof(long));
    assert(p);
    for (size_t i = 0; i < size; i++) {
        p[i] = 0;
    }
    return p;
}
```

(c) _____

_____

```
void poke(char *p, char c) {
    *p = c;
}
```

(d) _____

_____

```
char peek(char *p) {
    return *p;
}
```

(e) _____

_____

```
int abs_dynamic(int x) {
    int *pos_ptr = malloc(sizeof(int));
    int result;
    assert(pos_ptr);
    if (x >= 0) {
        *pos_ptr = x;
        result = *pos_ptr;
        free(pos_ptr);
    }
    if (x <= 0) {
        *pos_ptr = -x;
        result = *pos_ptr;
        free(pos_ptr);
    }
    return result;
}
```

Here are some reminders about some C functions that appear in the question. `assert` (which is technically a macro) takes as an argument a boolean condition that is supposed to be true. If the condition is true nothing happens, and if the condition is false the program immediately stops with an error message. `malloc` takes a number of bytes as an argument, and allocates that much memory, returning a pointer to the allocated memory. The pointer returned by `malloc` should be passed to `free` when the program is done with it. `strcpy` and `strcat` both copy a string into a destination buffer in their first argument. The difference between them is that `strcpy` overwrites the buffer from the beginning, while `strcat` performs concatenation by copying its second argument after the end of the string already in the destination buffer.

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) ____ Library function to execute a string with a shell

   (b) ____ System call to change user and group associated with a file

   (c) ____ Exempt from all discretionary access-control checks

   (d) ____ Virtual machine underneath a normal kernel

   (e) ____ Allow-list-style mechanism to stop control-flow hijacking

   (f) ____ x86-64 register pointing to the beginning of a stack frame

   (g) ____ CPU mode where all memory is accessible

   (h) ____ Specifying which inputs constitute an attack

   (i) ____ Subject to buffer overflow and format string bugs

   (j) ____ Stores CPU registers in a memory buffer

   (k) ____ Compilation mode where all code is position-independent

   (l) ____ Added to password hash to conceal equality

   (m) ____ Allows information to flow in only one direction

   (n) ____ A value which, if overwritten, indicates an attack

   (o) ____ x86-64 register pointing to the top of the stack

   (p) ____ Architecture vulnerability related to, e.g., branch prediction

   (q) ____ An isolated environment for untrusted code

   (r) ____ CPU mode where page tables cannot be changed

   (s) ____ Trusted Computer System Evaluation Criteria

   (t) ____ Point where false-positive and false-negative rates are equal

   A. canary B. CFI C. `chown` D. data diode E. deny list F. EER G. hypervisor H. kernel mode I. Orange Book J. PIE K. `%rbp` L. `%rsp` M. salt N. sandbox O. `setjmp` P. Spectre Q. `sprintf(3)` R. superuser S. `system(3)` T. user mode