Computer Science 4271
Fall 2023
Midterm exam 1
October 10th, 2023
Time Limit: 75 minutes, 4:00pm-5:15pm

- 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 start. Don't put any of your answers on this page.
- This exam contains 5 pages (including this cover page) and 3 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 each question carefully before answering it. 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 within the available time and space.
- 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, to an aisle: _____

Sign and date: _____

Question	Points	Score
1	42	
2	28	
3	30	
Total:	100	

1. (42 points) Stack buffer overflow and an attempted defense

   Your co-worker Corrie likes programming in C, but sometimes has trouble with security, and you've been called in to explain a problem that has come up. Corrie's first mistake was writing a function that is vulnerable to a stack buffer overflow attack. Corrie then had the idea of blocking such attacks by keeping track of what the return address is supposed to be, and aborting execution if it has been modified. However, this defense idea turns out to not be very strong, because of how the code has been compiled.

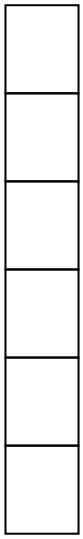
   Here is a cut-down version of Corrie's function, showing just the dangerous operation and the attempted defense:

```
void func(char *s, size_t sz) {
    void *orig_ret = __builtin_return_address(0);
    char buf[16];
    memcpy(buf, s, sz);
    if (__builtin_return_address(0) != orig_ret)
        abort();
}
```

   The argument `s` to the function is a pointer to some binary data, and `sz` is the size of that data in bytes. The function `__builtin_return_address` is a GCC extension which when called with the argument 0 evaluates to the current function's return address, as stored on the stack. The function `abort` is a standard one that immediately halts the program's execution: if it is called then `func` will never return.

   (a) On the left below is the assembly code for the function. In the middle is a diagram showing the stack frame used by the function, split into 8-byte boxes. Each box is labeled by its location relative to the value of the `%rbp` register, specifically the value that register has after the `push %rbp` instruction and before the matching `pop %rbp`. The final column has letters describing different data stored in the stack frame. Fill each box in the middle column with the letter from the right that describes its contents. Each letter should be used at most once, except perhaps F ("unused").

```
func:
  push   %rbp
  mov    %rsp, %rbp
  sub    $0x20, %rsp
  mov    %rsi, %rdx
  mov    0x8(%rbp), %rax
  mov    %rax, -0x8(%rbp)
  lea    -0x20(%rbp), %rax
  mov    %rdi, %rsi
  mov    %rax, %rdi
  call   memcpy
  mov    0x8(%rbp), %rax
  cmp    %rax, -0x8(%rbp)
  jne    fail
  mov    %rbp, %rsp
  pop    %rbp
  ret
fail:
  call   abort
```



A. `buf[8]` through `buf[15]`

B. `orig_ret` variable

C. `func`'s return address

D. `buf[0]` through `buf[7]`

E. saved `%rbp`

F. unused

(b) Corrie tested the defense by supplying the following input (shown in the syntax of a C string; the size is 48 bytes):

`AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE\xad\x11\x40\0\0\0\0\0`

Corrie observes that this replaces the function's normal return address, `0x4012dd`, with the shellcode address `0x4011ad`. But before the function jumps to the shellcode, the comparison fails and it calls `abort`. To Corrie, that sounds like the defense is working.

However, there is a detail missing in the description above, which actually shows that the defense is not working quite as intended. Explain what is happening.

(c) Even more direct evidence that the defense is flawed is that one and only one of the following inputs (same syntax and length as before) is a working attack that does cause the shellcode to be executed. Circle the letter of the working attack.

A. `AAAAAAAABBBBBBBBCCCCCCCC0x4012ddEEEEEEEE0x4011ad`

B. `AAAAAAAABBBBBBBBCCCCCCCC\xad\x11\x40\0\0\0\0\0EEEEEEEE\xad\x11\x40\0\0\0\0\0`

C. `AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD\xad\x11\x40\0\0\0\0\0\xad\x11\x40\0\0\0\0\0`

D. `AAAAAAAABBBBBBBBCCCCCCCC\xad\x11\x40\0\0\0\0\0\xad\x11\x40\0\0\0\0\0FFFFFFFF`

E. `AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE\xad\x11\x40\0\0\0\0\0`

F. `AAAAAAAABBBBBBBBCCCCCCCC\xdd\x12\x40\0\0\0\0\0EEEEEEEE\xad\x11\x40\0\0\0\0\0`

G. `AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFF`

(d) You might say that Corrie was unlucky in the decisions the compiler made. How might a C compiler have compiled this function differently in a way that would prevent the kind of attack we've described here?

2. (28 points) Multiple choice. Each question has only one correct answer: circle its letter.

   (a) We've mentioned that the most commonly used no-op instruction for NOP sleds on x86-64 is the one-byte instruction `0x90`. But suppose that instead you used the two-byte instruction `0x66 0x90`, which also does nothing. You make a 1000 byte NOP sled consisting of 500 copies of `0x66 0x90`, followed by your normal shellcode. If an attack jumps to a random byte offset in the NOP sled, the chances that the shellcode will execute are about:
   A. 0 B. 25% C. 33% D. 50% E. 100%

   (b) Suppose that each of the values shown in hex below is multiplied by 48 (`0x00000030`), using the rules of 32-bit multiplication (like an int in C). Which one overflows to give the value 96 (`0x00000060`)?
   A. `0x20202020` B. `0x7fffffff` C. `0x10000001` D. `0x00000001` E. `0x20000002`

   (c) Dowd et al. recommend, among other things, a series of code auditing strategies they label DG1 through DG4. What does "DG" stand for in these names?
   A. density gradient
   B. Data General
   C. design generalization
   D. data and generation
   E. defense grading

   (d) This `printf` format specifier can't be used to modify data, but it could be used in a format string attack to reveal information or to make the program crash (e.g. with a segfault):
   A. `%s` B. `%x` C. `%ld` D. `%c` E. `%d`

   (e) Because you press down while writing with a pencil or pen on a pad of paper, you might leave subtle indentations in the shape of your writing on the second page, even after you have taken the top page you've written on away. What kind of security problem is this?
   A. spoofing B. tampering C. repudiation D. information disclosure E. denial of service

   (f) One obvious reason not to use a pencil or another easily-erasable writing instrument for important documents is that it would be easy for someone with momentary access to the documents to change their contents. What kind of security problem is this?
   A. spoofing B. tampering C. repudiation D. information disclosure E. denial of service

   (g) As another consequence of ease of modification, suppose you sign and date a contract in pencil. If you later want to deny the applicability of the contract to an event on a specific date, you might claim that you'd signed it later, and that the date had been modified. What kind of security problem is this?
   A. spoofing B. tampering C. repudiation D. information disclosure E. denial of service

3. (30 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

(a) ____ Another phrase describing code auditing

(b) ____ Modeled as the product of the loss from an event times its probability

(c) ____ Code that executes the same at any memory address

(d) ____ Filling stack space with no-op ROP gadgets

(e) ____ Starting point for back tracing in code auditing

(f) ____ The register used for the first function argument on x86-64

(g) ____ Modifying code to run at a different address

(h) ____ A C function similar to `goto` across functions

(i) ____ The register pointing to the lowest-addressed stack location

(j) ____ A fuzzing tool named after a breed of rabbits

(k) ____ Position-independence applied to the main program

(l) ____ The "frame pointer" to the high end of a stack frame

(m) ____ Components that record information without initiating actions

(n) ____ Gaining the ability to do things you shouldn't be able to

(o) ____ When a C compiler can make your program do anything

A. AFL B. application review C. candidate point D. data store E. elevation of privilege F. `longjmp` G. PIC H. PIE I. `%rbp` J. `%rdi` K. relocation L. ret2pop M. risk N. `%rsp` O. undefined behavior