Computer Science 4271
Fall 2023
Midterm exam 1 (solutions)
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 8 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.

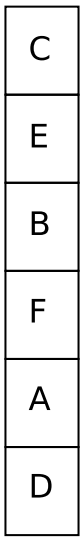
   Here is a cut-down version of Corrie's function, showing just the dangerous operation and the attempted defense:

```
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

A lot of this layout is similar to examples we've seen before, and you could also work backwards from the later attack information to guess what's going on here. But the intended/best way to figure out the layout is to look at the operations in the assembly code and what locations they use. You should see a number of operands that look similar to labels in the stack frame picture. The buffer `buf` *is passed as the first argument to* `memcpy`*, so you can find it by working backwards from the call to* `memcpy` *and seeing what's in the first argument register* `%rdi`*. Specifically, you can see that* `%rdi` *is copied from* `%rax`*, and in turn* `%rax` *is computed using* `lea` *as* `-0x20(%rbp)`*.(This is an* `lea` *instruction because it's computing the address, not loading a value from the stack.) This means that* `-0x20(%rbp)` *is the starting address of the array (D), and the array grows upward from there (A). It is a standard convention that the frame pointer points at the old saved frame pointer, so E is at* `(%rbp)`*,but you can also see that by observing that the code copies the stack pointer to* `%rbp` *right after pushing the old* `%rbp` *value. The code that pushes the return address isn't in this snippet (it's the call instruction), but the* `ret` *instruction pops the return address, so you can see that it (C) needs to be above E (pushed earlier, popped later). The two places that access the* `orig_ret` *variable both also access the return address, so those two values have to be* `0x8(%rbp)` *and* `-0x8(%rbp)` *in some order. The comparison is symmetric, but the first use copies from the return address to* `orig_ret`*, so you can distinguish them based on the order of operations. The remaining location,* `0x10(%rbp)` *is not accessed by any of the code, so it gets F.*

(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.

Carrie is correct that the bytes `\xad\x11\x40\0\0\0\0\0` *(a total of 8 bytes, including most-significant zero bytes) are overwriting the return address. But because of the stack layout (as illustrated in the previous question), other data on the stack is also getting overwritten. In particular, the string of bytes DDDDDDDD will overwrite the location where the* `orig_ret` *variable is stored. In hex this is 0x444444444444444, so it is still different from the from the overwritten return address, and the attack is still getting detected. But instead of comparing the overwritten return address with the correct return address, the code is comparing it with a value controlled by the attacker.*

(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. `AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFF`

The way to make the attack effective is to overwrite the saved return address in `orig_ret` *with the same value as we're using to overwrite the return address, so that the security check will pass but the program will still jump to the new address. This is what the input B does. D and C also use the same values, but they won't work because they overwrite the wrong locations. F has the right layout, but it overwrites the* `orig_ret` *variable with the correct return address, which ensures that the security checks works correctly. E is the same input mentioned in the previous part, which fails as described before, and A is similar to D but tries to overwrite addresses with ASCII hex characters, which does not give the correct values.*

(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?

The best answers we were hoping for here were decisions that a compiler makes on its own (e.g., not directly controlled by command-line options), and which are not normally made for security reasons but would happen to make a difference here. Two such examples would be deciding to keep the `orig_ret` *variable in a register, rather than on the stack, or putting the stack variables in a different order, with* `buf`*'s address higher than* `orig_ret`*'s, so that a forward overflow from* `buf` *wouldn't affect* `orig_ret`*. (In fact, we encountered both of these behaviors when trying to create this example with GCC.) We also have partial credit for other compiler-based defenses, but these wouldn't be as good an answer if the decision about turning them on or off was made by the user in selecting compiler options, and some defenses aren't really relevant to this problem. For instance enabling ASLR doesn't randomize the layout of stack frames, so it wouldn't have interfered with the overwrite. (The question doesn't mention how the attacker determines the shellcode address, but in some cases ASLR would randomize it.)*

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%**

   A NOP sled based on a two-byte NOP instruction would always be at least 50% effective in the metric the question asks about, because the NOP sled will definitely work as intended if execution starts at the beginning of one of the NOP instructions. If the instruction starts at the second byte of a two-byte NOP instruction, the byte sequence might be interpreted differently so in general it might fail. This might be part of why D was the most common wrong answer. However in this case we can see what will happen if execution starts at the second byte of the `66 90` *instruction, because the question reminds us that* `90` *is just a one-byte NOP instruction. It will still just be a NOP instruction, and then execution will re-align with the two-byte NOP instructions. Thus for this particular two-byte instruction, any location will still work.*

   (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`

   96 is 2 times 48, which may be even easier to see in hex. So to get the low part of the result right, we want the low part of the value we're multiplying to also be `0x2`*. If it were* `0x00000002`*, that would also produce the desired result, but it wouldn't be an overflow. Because 48 is a multiple of 16, multiplying by 48 effectively includes shifting left by 4 bit positions or one hex digit, so any value in the most significant hex digit will be shifted away. That's why multiplying by* `0x20000002` *overflows to the same result as multiplying by* `0x00000002`*.*

   (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

   "Design generalization (DG) strategies focus on identifying logic and design vulnerabilities by reviewing the implementation and inferring higher-level design abstractions." (Dowd, p. 34–35 in our PDF).

(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`

All five of these format specifiers can't be used to modify data, and all five could be used to reveal information. `%x`*,* `%ld`*,* `%c`*, and* `%d` *all reveal data directly from the stack by treating it like an integer-family data type.* `%s` *is a little bit different because it treats the value it reads from the stack as a character pointer and prints the characters it points at up to a null byte. This is a kind of revealing information that could be useful in other circumstances, but it will also make the program crash with a segfault if the value read from the stack is not a valid pointer.*

(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

This technique gives the attacker access to information that we wanted them not to know, so it is information disclosure.

(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

If an attacker changes the information in a document when we would not want them to, that is a tampering attack.

(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

An attacker later denying an action they took or commitment they made in the past is repudiation.

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) __**B**__ Another phrase describing code auditing

   (b) __**M**__ Modeled as the product of the loss from an event times its probability

   (c) __**G**__ Code that executes the same at any memory address

   (d) __**L**__ Filling stack space with no-op ROP gadgets

   (e) __**C**__ Starting point for back tracing in code auditing

   (f) __**J**__ The register used for the first function argument on x86-64

   (g) __**K**__ Modifying code to run at a different address

   (h) __**F**__ A C function similar to `goto` across functions

   (i) __**N**__ The register pointing to the lowest-addressed stack location

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

   (k) __**H**__ Position-independence applied to the main program

   (l) __**I**__ The "frame pointer" to the high end of a stack frame

   (m) __**D**__ Components that record information without initiating actions

   (n) __**E**__ Gaining the ability to do things you shouldn't be able to

   (o) __**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