Computer Science 5271
Fall 2019
Midterm exam (solutions)
October 21st, 2019
Time Limit: 75 minutes, 1:00pm-2:15pm

- This exam contains 11 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 2:15pm. 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	30	
2	24	
3	26	
4	20	
Total:	100	

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

   (a) This character is used as part of a printf format specifier to control what argument a value is taken from:

   A. `@` **B.** `$` C. `0` D. `s` E. `-`

   This is a `printf` *feature that was not used in Exercise set 1, but it was used in BCMTA for a benign purpose, and it's also useful in format string attacks if you want to skip a lot of argument positions.* `0` *and* `-` *are other format specifier modifiers, and* `s` *is another format specifier, but they all have different purposes.* `@` *does not have a special meaning for* `printf`*.*

   (b) The pathname passed to the Unix open system call cannot contain this character:

   A. `/` **B.** `\0` C. `\` D. `*` E. space

   At the level of the kernel interface, Unix is very permissive about what characters may be used in filenames. The only special byte values are null and forward slash (`/`*): a slash separates levels of directory and a null terminates a pathname string. You can specify a relative or absolute path in* `open` *by including slashes, which is what makes the argument to* `open` *a "pathname" instead of a "filename'. Space,* `*`*, and* `\` *all have special meanings to Unix shells so you have to take special steps to include them in filenames, but all are allowed.*

   (c) This security design idea is used in both qmail and Android:

   A. Using Unix UIDs to isolate code instead of human users
   B. Using C++'s `std::string` class exclusively
   C. Requiring add-on software to be cryptographically signed
   D. Sandboxing code from the Internet using SFI
   E. Having only a single developer

   Traditionally Unix UIDs identify users, but in qmail they are used to separate different parts of qmail, like `qmailr` *for* `qmail-remote`*. In Android, each app is assigned its own UID. Neither qmail nor Android use* `std::string` *exclusively, in fact neither is primarily written in C++. Android apps are signed, but qmail does not have an analogous third-party extension mechanism. Neither qmail nor Android use SFI; the most similar feature is that Android apps are primarily written in Java. qmail had only one main developer, but Android is a commercial project with a large development team.*

   (d) Which of these values would commonly **not** change under ASLR?

   A. The address of an environment variable
   B. The address of `main`'s stack frame
   C. The relative distance between `main` and `printf`
   D. The relative distance between a return address and a local variable
   E. The address of `printf`

   An environment variable and the stack frame of `main` *are both part of the stack, and so move when the base address of the stack is randomized.* `printf` *is usually in shared library that is randomized separately from the main program (if the main program is randomized at all), so both the location of* `printf` *and the distance between* `main` *and* `printf` *vary based on the library's randomized location. But a return address and a local variable are both located within the stack, and the layout of a stack frame is chosen by the compiler, so their relative distance is not changed by ASLR.*

(e) "Heartbleed" was the name of a high-profile vulnerability disclosed in 2014. It was caused by faulty bounds checking, but because the unsafe access was a read instead of a write, the only kind of security policy that was directly violated was:

A. confidentiality B. availability C. authentication D. integrity

Reading data that should not be read is a violation of a confidentiality policy.

(f) Under a CFI implementation with two equivalence classes for calls and returns, and no shadow stack, an attacker could still redirect a function return to:

A. A gadget that starts in the middle of an intended instruction

B. Shellcode in an environment variable

C. A call-preceded gadget

D. The `system` function in the C library, always

E. The `system` function in the C library, but only if its address was taken

In a two-equivalence-class CFI implementation, indirect calls can only go to legal call targets, and returns can only go to legal targets for returns, namely locations right after call sites. A gadget in the middle of an intended instruction or shellcode in an environment variable would never be legal control-flow targets at all, so they would be blocked by any kind of CFI (shellcode in an environment variable would also be blocked by W$\oplus$X). A coarse-grained CFI system might allow a library function like `system` *to be a legal target, especially if it is called or has its address taken in the program, but a two-equivalence-class system would still prevent a return to* `system`*, which would be mixing a return with a call target. A call-preceded gadget is one that starts at a location after a call instruction that would be a legal place for some return instruction to target, so its address would be part of the legal return address equivalence class.*

(g) Password capabilities are similar to passwords in that:

A. They are chosen by users

B. To be secure, they must have high entropy

C. They are gradually being replaced by fingerprints

D. They are vulnerable to dictionary attacks

E. To be secure, they must be changed frequently

Password capabilities are an OS-internal mechanism that represent the ability to access an OS resource. They are not chosen by users or based on English words, and it doesn't make sense to replace them with a biometric because they are not intended to identify users in the first place. Because a password capability does not need to be remembered by a human, it can be chosen long enough that even a long brute-force/"dictionary" attack is impractical, so it is not important to change it frequently. But a password capability would be insecure if had low enough entropy that an adversary could guess it, and since the checking of a password capability might be within a single system, the rate of attack might be relatively fast.

(h) One feature commonly added to high-security operating systems is tamper-proof logging. Which of Saltzer and Schroeder's design principles is this an instance of?

A. compromise recording
B. separation of privilege
C. confidential reservation
D. separation of powers
E. separation of duty

Logging is useful for security as a kind of compromise recording. If this is going to be useful a security mechanism, it must not be possible for an attacker who has compromised the system to also tamper with the logs to remove the evidence of the compromise; the difficulty of making such logging tamper-proof is part of why this principle is not always applicable. "Confidential reservation" starts with the same letters but is not one of Saltzer and Schroeder's principles. Separate of privilege is another of Saltzer and Schroeder's principles, and separation of duty is another name for a kind of separation of privilege between users. Separation of powers is a concept in politics which is only more distantly related to separation of privilege.

(i) Which of these CPU features does **not** lead to transient execution?

A. single-instruction multiple-data (SIMD) instructions
B. branch target prediction
C. return stack buffer usage
D. delayed memory exceptions
E. branch direction prediction

Delayed memory exceptions are the cause of transient execution in the Meltdown attack, and branch direction prediction and branch target prediction are the two most common causes of transient execution in the Spectre attack. The return-stack buffer is a specialized hardware feature used to predict the targets of return addresses. It is thus used for a kind of target prediction, and the survey paper mentions that it has also been used in Spectre-like attacks. SIMD instructions allow a processor to do several arithmetic operations at once, but doing all the operations in parallel is part of the architectural semantics, and does not require transient execution.

(j) At the University of Minnesota, you type in the same username and password to log into CSE Labs workstations, to access the WiFi network, and to view library resources off campus. This is an example of:

A. two-factor authentication

B. biometric authentication

C. economy of mechanism

D. centralized authentication

E. single sign-on

A password is not biometric, and a password on its own is just a single factor. This isn't a good example of economy of mechanism because the different systems are separately implemented. The best answer is centralized authentication: there is a centralized database of usernames and passwords that all these otherwise-separate systems connect to. (You might say that centralization is a description of the implementation, and we don't actually now how all of these systems are implemented, but the fact that the same username and password are used is a sign that part of the mechanism is centralized.) Single sign-on is a description of the user experience, and I would the three examples listed in the question are not single sign-on: you would have to type the same username and password three times to access WiFi, use a library resource in your browser, and connect to a CSE Labs workstation. However the U's centralized authentication does provide single sign-on among resources that do web-based authentication, which include MyU and Canvas as well as the library proxy. So we also gave partial credit for the answer of "single sign-on".

2. (24 points) Social media reliability relations

   A commonly-raised concern about social media is that it is sometimes used to propagate unreliable information. Inspired by the use of lattices to define integrity policies in multi-level secure systems, your friend Parker is considering applying a similar mechanism to deal with unreliable information in Twitter.

   (a) Parker's basic idea is to define an ordering relation $\sqsubseteq$ based on reliability. $B \sqsubseteq A$ will hold when user A is more reliable than B; only if this is the case will B be allowed to read A's tweets. On the left are some mathematical properties that the relation $\sqsubseteq$ might have. Match them with the intuitive descriptions on the right of what the properties mean in this context: each one is used exactly once.

   A. If $A \sqsubseteq B$ and $B \sqsubseteq C$, then $A \sqsubseteq C$

   B. $A \sqsubseteq A$

   C. If $A \sqsubseteq B$ and $B \sqsubseteq A$, then $A = B$

   D. Either $A \sqsubseteq B$ or $B \sqsubseteq A$

   __**B**__ Everyone can read their own tweets

   __**C**__ If two users are each at least as reliable as the other, they are equally reliable

   __**D**__ For any two people, at least one can read the other's tweets

   __**A**__ If you can read a retweet, you can also read the original tweet

   (b) Keeping with the spirit of social networks, Parker has decided that reliability will be defined based on popularity, specifically by a pair of non-negative integers (t, i) where t is the number of Twitter followers a user has, and i is their number of Instagram followers. (t or i is also 0 if the user doesn't have Twitter or Instagram account at all.) But there was still some disagreement about the exact definition.

   Leslie believes that Twitter is a better indication of reliability than Instagram, and proposed the following definition:

$$(t_1, i_1) \sqsubseteq_L (t_2, i_2) \iff t_1 < t_2 \vee (t_1 = t_2 \wedge i_1 \leq i_2)$$

   Does Leslie's definition satisfy all four properties A-D mentioned above? If not, choose one property and give an example of a situation in which it does not hold.

   All four of the properties A–D hold for $\sqsubseteq_L$. In other terminology, this means that $\sqsubseteq_L$ is a total order.

   (c) Chris thought it was more important for the definition to treat Twitter and Instagram equally, and proposed the following definition:

$$(t_1, i_1) \sqsubseteq_C (t_2, i_2) \iff t_1 \leq t_2 \wedge i_1 \leq i_2$$

   Does Chris's definition satisfy all four properties A-D mentioned above? If not, choose one property and give an example of a situation in which it does not hold.

   $\sqsubseteq_C$ satisfies A–C, but not D. One example of property D failing is that $(0, 10) \not\sqsubseteq_C (10, 0)$, but also $(10, 0) \not\sqsubseteq_C (0, 10)$. In other words, $\sqsubseteq_C$ is a partial order but not a total order. It has pairs which are incomparable, like $(0, 10)$ being incomparable with $(10, 0)$.

(d) Dakota liked Chris's suggestion, but wanted a definition that allowed more tweets to be read, and so proposed the following:

$$(t_1, i_1) \sqsubseteq_D (t_2, i_2) \iff t_1 \leq t_2 \vee i_1 \leq i_2$$

(This is the same as Chris's, but with "or" instead of "and".) Does Dakota's definition satisfy all four properties A–D mentioned above? If not, choose one property and give an example of a situation in which it does not hold.

Properties A and C both fail for $\sqsubseteq_D$. *An example of the failure of A is that* $(4,3) \sqsubseteq_D (1,4)$ *(because* $3 \leq 4$*) and* $(1,4) \sqsubseteq_D (2,1)$ *(because* $1 \leq 2$*), but* $(4,3) \not\sqsubseteq_D (2,1)$.

As an example of the failure of C you can use the same kind of pairs that were examples of failures of D in the previous part. For instance $(10,20) \sqsubseteq_D (20,10)$ *and* $(20,10) \sqsubseteq_D (10,20)$*, but* $(10,20) \neq (20,10)$. *A more open-minded reader might also ask why we can't just get out of that counterexample by claiming that* $(20,10) = (10,20)$ *for the purposes of the ordering. But it doesn't make sense to consider those equal because they are ordered differently with respect to other elements. For instance* $(25,15) \sqsubseteq_D (10,20)$*, because* $15 \leq 20$*, but* $(25,15) \not\sqsubseteq_D (20,10)$.

(e) Which of the definitions $\sqsubseteq_L$, $\sqsubseteq_C$, and/or $\sqsubseteq_D$, form(s) a lattice?

Both $\sqsubseteq_L$ *and* $\sqsubseteq_C$ *can form a lattice when matching meet and join operators are supplied, but* $\sqsubseteq_D$ *cannot form a lattice, because a lattice is a kind of partial order, and a partial order has to satisfy all of the properties A–C. We saw that a lot of students only mentioned one of the definitions in their answer here, perhaps because we were not explicit enough (the "and/or" and "(s)" were intended to be signs) that we wanted to know the whole list of definitions that form lattices. But mentioning one of the two correct answers only lost you one point.*

(f) Before Parker and his friends visit California to talk to venture capitalists, is there any other problem with or objection to their idea they should consider?

There are many possible problems, and there wasn't any single one we wanted over all others. On a technical side, these lattices have a lot of levels, so you probably wouldn't want to represent them all explicitly. From a practical point of view, this system would make it hard for new users to get started using the systems, since at first very few people can read their messages, but without that it's hard to get followers. Having many followers may not be a good stand-in for reliability: someone might be popular without having reliable information. This mechanism requires people who have many followers to get more of their information outside the social network, but there's no guarantee that the other information sources that substitute for the social network will be better. From a business standpoint, talking to VCs suggests that Parker and friends want to make money from this idea, but it's not obvious where the profit opportunity is. If Parker and friends want to start a new social network, it may be hard to get users if it starts out small and more reliable information may not be enough to get users to switch. If Parker and friends are hoping that Twitter or Instagram will buy this idea, it's not obvious how those companies would increase revenue by making information more reliable.

3. (26 points) Return-to-libc attack.

   You are trying to carry out an attack against a Linux/x86-32 program that has a buffer overflow vulnerability. In your preliminary research, you've found that the vulnerability is completely permissive as to the data used in the overflow (even `\0` bytes are OK), and ASLR is disabled. But the bad news is that it only lets you write 44 bytes into a 32-byte buffer, which is not enough to overwrite a return address.

   Your backup plan is to carry out a return-to-libc attack, to let you make the program do the equivalent of `system("/tmp/evil")` using code that already exists in the binary. (You have already prepared the script `/tmp/evil`.) But you have to figure out how to make that happen by overwriting just the data that the overflow reaches. Below we've shown the C and assembly code for the relevant parts of the vulnerable program. On the next page, we've shown a picture of the area of memory that holds function `g`'s stack frame. For the locations that you control, there are blank spaces for you to fill in the data you want to go in those locations. Each blank space represents one byte: fill it in with either a single ASCII character like `A` or two hex digits like `ff`. You may leave a blank empty if it is not needed for your attack.

   The address of the function `system` is `0x0804f170`. It's not too important what the program does after calling `system`, but if you'd like to have it call `exit`, the address of that function is `0x804e670`.

   The blank spaces for the bytes are in order of increasing address left to right. So be careful about the order in which you write things, for instance depending on if they are a string or an address. (x86 is little-endian, putting the least-significant byte of a word at the lowest address.)

```
unsigned char input[44];
void f(char *str) {
    int len =
      parse(str, input, sizeof(input));
    g(input, len);
}

void g(unsigned char *data, int len) {
    unsigned char buf[32];
    memcpy(buf, data, len); /* oops */
    return;
}
```

```
f:
   /* ... */
   call   g
   add    $0x10,%esp
   mov    %ebp, %esp
   pop    %ebp
   ret

g:
   push   %ebp
   mov    %esp,%ebp
   sub    $0x28,%esp
   mov    0xc(%ebp),%eax
   sub    $0x4,%esp
   push   %eax
   pushl  0x8(%ebp)
   lea    -0x28(%ebp),%eax
   push   %eax
   call   memcpy
   add    $0x10,%esp
   mov    %ebp, %esp
   pop    %ebp
   ret
```

You'll that a return-to-libc attack is an older and simpler kind of code reuse attack. Like more sophisticated ROP attacks, return-to-libc doesn't use any shellcode: the attack is carried out only using code that is already present inside the vulnerable program. Most specifically, the phrase "return-to-libc" refers to hijacking a function return so that it goes not to the normal return site but to the entry point of an attacker-useful function in the C library. The `system` *function used in this question is a classic target because of its shellcode-like functionality: given a string argument represented as a character pointer, it passes the contents of the string to the shell. We already did the part of figuring out what string he attacker wants to pass, but that string is probably not already present in the binary, so it should be part of the attack payload.*

The one factor that makes this question different from the most classic return-to-libc is that the overflow is not long enough to overwrite the return address directly. However the overwrite can replace a saved value of the frame pointer `%ebp` *on the stack, which is enough to replace a return because it changes the way the program reads data from the stack. Note that even though the overflow happens inside* `g`*, the saved frame pointer is used in* `f`*, so it is* `f`*'s return instruction that should get hijacked; this is why we included the code for the last few instructions of* `f`*. A good way of thinking about how this attack should work is that we want to create a fake section of stack that we are going to trick* `f` *into using, by changing its* `%ebp` *value. The values in this fake stack area should have the same offsets from each other that the code in* `f` *is expecting, but the area will be inside the attacker-controller overflow.*

The two most important values for the attack are the address of `system`*, and the pointer to the attacker-supplied command string. You have to be careful in thinking about their positions because there's another change of perspective that happens when we switch to executing* `system`*:* `f` *treats the address of* `system` *like its return address, but then once* `system` *starts executing, it thinks that it has been called.* `system` *will look for its argument on the stack, but* `system` *also expects* `system`*'s return address to be on the stack. Because the attack "called"* `system` *with a return instead of a normal call, that return address isn't set up automatically by the instructions; you need to think about it in constructing the fake stack. The value of* `system`*'s return isn't going to matter to the success of the attack, but we gave you the address of* `exit` *as a hint that you could use it as the "return" address of* `system`*, as a very simple kind of chained return-to-libc attack. However it is important for the success of the attack that the pointer to the command string be two words above the address of* `system`*, because the location in between is where* `system` *expects to find its own return address.*

Where should the overwritten `%ebp` *point, relative to our saved stack frame? The convention for saved* `%ebp`*s is that they each point at the calling function's saved* `%ebp`*, so the fake* `%ebp` *should point at the place where* `f`*'s saved* `%ebp` *would go, one word below* `f`*'s return address. As with* `system`*'s return address, the value is not so important because it will only be used after the attack, but the layout is important.*

Because the blank spots correspond to bytes in order of increasing address, but x86 is little-endian, all the 32-bit addresses need to be written with their bytes reversed. On the other hand, the string `/tmp/evil` *still has its bytes in increasing address order, so it should be written left to right, and then up. It's also necessary to include a null terminator for the command string, so* `system` *knows where it ends. One final point to keep in mind is that it is better to have the command string at a higher address and the fake stack at a lower address, because* `system` *itself will use the stack below its argument and return address for its own purposes, and we don't want* `system` *to overwrite the command string before using it.*

Address	Original purpose	Overwrite with
`0xffffc3d4`:	`g` arg 2 (`len`)	
`0xffffc3d0`:	`g` arg 1 (`data`)	
`0xffffc3cc`:	`g`→`f` return address	
`0xffffc3c8`:	saved `%ebp`	`a0` `c3` `ff` `ff`
`0xffffc3c4`:	unused	___ ___ ___ ___
`0xffffc3c0`:	unused	___ ___ ___ ___
`0xffffc3bc`:	`buf[0x1c]` – `buf[0x1f]`	___ ___ ___ ___
`0xffffc3b8`:	`buf[0x18]` – `buf[0x1b]`	`l` `00` ___ ___
`0xffffc3b4`:	`buf[0x14]` – `buf[0x17]`	`/` `e` `v` `i`
`0xffffc3b0`:	`buf[0x10]` – `buf[0x13]`	`/` `t` `m` `p`
`0xffffc3ac`:	`buf[0x0c]` – `buf[0x0f]`	`b0` `c3` `ff` `ff`
`0xffffc3a8`:	`buf[0x08]` – `buf[0x0b]`	`70` `e6` `04` `08`
`0xffffc3a4`:	`buf[0x04]` – `buf[0x07]`	`70` `f1` `04` `08`
`0xffffc3a0`:	`buf[0x00]` – `buf[0x03]`	___ ___ ___ ___

Some more hints:

- After returning from `memcpy`, `%esp` contains `0xffffc3a0` and `%ebp` contains `0xffffc3c8`.
- In AT&T syntax, the destination is the final operand, so for instance `mov %eax, %ebx` copies the contents of `%eax` into `%ebx`.
- The arguments to functions on Linux/x86-32 are passed on the stack, with the first argument at the lowest address. For instance, the relative position of the arguments and return address shown for `g` is also the same for other functions.
- `system` will need to create its own stack frame, so make sure that doesn't overwrite anything needed for your attack.
- You can assume that all of the stack addresses will be the same each time the program runs.

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) __**C**__ Common Orange Book level for enhanced Unix variants

   (b) __**E**__ Uses ambient authority for undesirable actions

   (c) __**K**__ Trusted code that checks all sensitive operations

   (d) __**S**__ A scripting language based largely on strings

   (e) __**P**__ System call used to drop privileges

   (f) __**J**__ Used to implement debugging and system call interposition

   (g) __**O**__ An isolated environment for untrusted code

   (h) __**H**__ Library function to reload register state

   (i) __**Q**__ Used on directories with multiple independent users

   (j) __**B**__ The absence of a connection between systems

   (k) __**I**__ An attack against kernel isolation using transient execution

   (l) __**A**__ A fuzzing tool that incorporates coverage feedback

   (m) __**T**__ A number identifying a subject in Unix access control

   (n) __**R**__ Will abort instead of overflowing a buffer

   (o) __**G**__ False positive and negative rate in a balanced configuration

   (p) __**F**__ A common mistake in dynamic memory management

   (q) __**M**__ Like a NOP sled for ROP

   (r) __**D**__ A filesystem-only isolation mechanism

   (s) __**L**__ An attack that is possible without an account

   (t) __**N**__ Used to prevent precompuation of password hashes

A. AFL B. air gap C. C2 D. `chroot` E. confused deputy F. double free G. EER H. `longjmp` I. Meltdown J. `ptrace` K. reference monitor L. remote exploit M. ret2pop N. salt O. sandbox P. `setresuid` Q. sticky bit R. `strcat_s` S. Tcl T. UID