Popa & Wagner
Spring 2020

CS 161
Computer Security

Midterm

PRINT your name: ____________________________, ____________________________
(last) (first)

I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in partial or complete loss of credit.

SIGN your name: ____

PRINT your SID: ____

Name of the person sitting to your left: _________________________

Name of the person sitting to your right: _________________________

You may consult **one** double-sided, handwritten sheet of paper of notes. You may not consult other notes or textbooks. Calculators, computers, and other electronic devices are not permitted.

Bubble every item completely. Avoid using checkmarks or Xs.
If you want to unselect an option, erase it completely and clearly.

For questions with **circular bubbles**, you may select only one choice.

○ Unselected option (completely unfilled)

● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

■ You can select

■ multiple squares (completely filled).

If you think a question is ambiguous, please come up to the front of the exam room to the TAs. We will not answer your question directly. If we agree that the question is ambiguous we will add clarifications to the document projected in the exam rooms.

There is an appendix on the last page of the exam, containing all signatures of all C functions used on this exam and a synopsis. Please do not remove this appendix from the exam.

You have 80 minutes. There are 7 questions of varying credit (100 points total).

Do not turn this page until your instructor tells you to do so.

Problem 1 ***Security Principles*** **(10 points)**

Select the best answer to each question.

(a) A company requires that employees change their work machines' passwords every 30 days, but many employees find memorizing a new password every month difficult, so they either write it down or make small changes to existing passwords. Which security principle does the company's policy violate?

○ Defense in depth ○ Ensure complete mediation

● Consider human factors ○ Fail-safe defaults

> **Solution:** Here is an article that discusses why password rotation should be phased out in practice, if you're interested in reading more.

(b) In the midst of a PG&E power outage, Carol downloads a simple mobile flashlight app. As soon as she clicks a button to turn on the flashlight, the app requests permissions to access her phone's geolocation, address book, and microphone. Which security principle does this violate?

○ Security is economics ● Least privilege

○ Separation of responsibility ○ Design in security from the start

> **Solution:** A flashlight application does not actually need these permissions in order to execute its functionality. It is over-permissioning its access to sensitive resources, violating the principle of least privilege.

(c) A private high school has 100 students, who each pay \$10,000 in tuition each year. The principal hires a CS 161 alum as a consultant, who discovers that the "My Finances" section of the website, which controls students' tuition, is vulnerable to a brute force attack. The consultant estimates an attacker could rent enough compute power with \$20 million to break the system, but tells the principal not to worry because of *which security principle?*

● Security is economics ○ Design in security from the start

○ Least privilege ○ Consider human factors

> **Solution:** The website handles \$1 million per year; not large enough that an attacker would have an incentive to spend \$20 million to steal it.

(d) The consultant notices that a single admin password provides access to all of the school's funds and advises the principal that this is dangerous. What principle would the consultant argue the school is violating?

○ Don't rely on security through obscurity ○ Design security in from the start

● Separation of responsibility ○ Fail-safe defaults

(e) Course staff at Stanford's CS155 accidentally released their project with solutions in it! In order to conceal what happened, they quickly re-released the project and didn't mention what had happened in the hope that no one would notice. This is an example of not following which security principle?

- ○ Security is economics
- ● Don't rely on security through obscurity
- ○ Separation of responsibility
- ○ Know your threat model
- ○ Least privilege
- ○ None of these

Solution: Uhh, can you guess where we got the idea for this question? Hint: It wasn't Stanford...

Problem 2 *Memory safety* **(14 points)**

(a) TRUE or FALSE: In the last question of Project 1, ASLR prevents the attacker from knowing the address of any instructions in memory.

○ TRUE ● FALSE

> **Solution:** In that question, the data and text segments were not randomized, so the attacker can find the address of program code and library code.

(b) TRUE or FALSE: An 8-byte stack canary is less secure than a 4-byte stack canary.

○ TRUE ● FALSE

> **Solution:** A 8-byte canary is no worse, and possibly better. It might be better because it is harder to guess, i.e., has more entropy.

(c) Format string vulnerabilities can allow the attacker to:

■ Read memory ■ Execute Shellcode

■ Write memory □ None of these

> **Solution:** When the attacker controls the format string, it is easy to read the stack with a variety of format specifiers. The %n identifier lets us write to certain parts of memory, and in some program lets us overwrite the RIP and execute shellcode.

(d) Which of the following memory safety hardening measures work by ensuring that all writeable regions in memory are non-executable, and all executable regions in memory are non-writeable?

□ ASLR ■ DEP (also known as WˆX or NX)

□ Stack canaries □ None of these

(e) Bear Systems hardens its code with both DEP (also known as WˆX or NX) and its own custom variant of ASLR. Normally, ASLR chooses a random offset for the stack and heap when the program starts running. Bear Systems modifies the compiler to choose a random offset when the program is compiled and hardcode this into the binary executable. Bear Systems ships the same executable to all of its customers. What is the effect of this modification to ASLR on security against memory safety exploits?

○ This modification makes security better.

○ This modification has no significant effect on security.

● This modification makes security worse.

Solution: This defeats the purpose of ASLR. Because the offset is hardcoded into the executable, it will be the same for all customers (i.e., the addresses will be the same for all customers). Thus, one customer can extract the offset from their copy of the executable, and then use it to infer the addresses used by other customers and attack other customers.

Problem 3 *Symmetric-key Cryptography* (16 points)

(a) TRUE or FALSE: AES-CBC mode requires both the sender and recipient to know the secret key and IV before communication begins.

○ TRUE ● FALSE

Solution: Only the secret key must be known by both parties beforehand. The IV is sent by the sender, as part of the ciphertext.

(b) TRUE or FALSE: AES-CTR mode with a non-repeating but predictable IV is IND-CPA secure.

● TRUE ○ FALSE

Solution: Knowing the IV doesn't help the attacker since the block cipher is applied first, and the output of the block cipher appears random.

(c) TRUE or FALSE: AES-CBC mode with a non-repeating but predictable IV is IND-CPA secure.

○ TRUE ● FALSE

Solution: See the Discussion Section 3 worksheet, Question 3(b).

(d) TRUE or FALSE: AES-ECB mode is IND-CPA secure if we prepend a random 16-byte value to the message before encryption and then encrypt the whole thing.

○ TRUE ● FALSE

Solution: The same attacks apply: if the first two blocks of the message are equal, then this fact will be detectable from the ciphertext, and so on. More generally, the attacker can just throw away the first 16 bytes of the ciphertext, and the result is the ordinary AES-ECB mode encryption of the message, which is vulnerable to all of the attacks mentioned in class.

Consider the following modified version of CTR mode:

$$C_i = \text{AES}_K(P_i \oplus (IV\|i))$$

where $\|$ denotes concatenation and $\oplus$ denotes bitwise xor. In other words, the xor occurs **before** applying the block cipher. As always, assume that the IV is sent with the ciphertext.

(e) TRUE or FALSE: If the IV is different for each message but predictable, this mode is IND-CPA secure.

○ TRUE ● FALSE

Solution: The attacks from Discussion Section 3 worksheet, Question 3(b), work here too.

A cryptography consultant suggests the following alternative mode:

$$C_i = \text{AES}_K(P_i) \oplus (IV \| i)$$

where $\|$ denotes concatenation and $\oplus$ denotes bitwise xor. In other words, the IV and counter are xored to the output of the block cipher. As always, assume that the IV is sent with the ciphertext.

(f) TRUE or FALSE: If the IV is chosen randomly for each message, the consultant's mode is IND-CPA secure.

○ TRUE ● FALSE

Solution: An attacker can compute $C_i' = C_i \oplus (IV \| i)$. Then $C_1', C_2', \ldots$ is an AES-ECB mode encryption of the plaintext, so we can apply all the attacks on ECB mode to this mode too.

Problem 4 ***Software Vulnerabilities*** **(11 points)**

Consider the following C code:

```c
1  // requires: s is a valid pointer, len <= size(s)
2  void f(char *s, size_t len) {
3      int i, j;
4      i=0; j=0;
5      while (j < len) {
6          // invariant: ???
7          while (s[j] == '<')
8              j++;
9          s[i] = s[j];
10         i++; j++;
11     }
12 }
```

(a) Assume we will only ever call `f` with arguments where `s` is a valid, non-null pointer to a buffer of length at least `len`, and that the attacker controls the data stored in `s`. Is this code memory-safe, under those conditions?

- ○ Yes, it is memory-safe
- ○ No, it could write past the end of the buffer
- ● No, it could read past the end of the buffer
- ○ No, it could write before the beginning of the buffer
- ○ No, it could read before the beginning of the buffer

Solution: This code can read past the end of the buffer if the data in the buffer ends with <, since there is no bounds check in the innermost while loop.

(b) If you selected "Yes", write a valid loop invariant for the place marked `???`. If you selected "No", write an example value for `s` and `len` that would trigger a memory safety violation.

Solution: `s` = >>>>, `len` = 4. Many other answers are possible. The common element to all of them is that `s[len-1] == '>'` (and if `len` is less than the size of the buffer, then all subsequent characters in the buffer must also be `'>'`).

As it happens, there is no integer overflow bug here (e.g., with `len == INT_MAX`+1), because C will cast both `j` and `len` to unsigned integer types before comparing them in line 5—but I could understand how you might think that one is possible. For that reason, I would also accept "No, it could read before the beginning of the buffer" or "No, it could write before the beginning of the buffer" in part (a) if you listed an example in part (b) where `len > INT_MAX`.

Problem 5 ***Public Key Encryption*** **(7 points)**

The El Gamal encryption scheme is reproduced below:

- **Key Generation**: public key = (g, h, p), where $h = g^k \pmod p$, private key = k
- **Encryption**: $c = (c_1, c_2) = (g^r \bmod p, m \times h^r \bmod p)$, where r is randomly sampled from $\{1, \ldots, p-1\}$.
- **Decryption**: $m = c_1^{-k} \times c_2 \pmod p$

Look at each scenario below and select the appropriate options.

(a) TRUE or FALSE: With El Gamal, it is not a problem if the adversary can learn the value of g somehow.

● TRUE ○ FALSE

Solution: g is part of the public key, so it is fine for it to be known to the public (including the adversary).

(b) TRUE or FALSE: With El Gamal, it is not a problem if the value r used during encryption is accidentally revealed after the encryption is complete.

○ TRUE ● FALSE

Solution: If the adversary learns r, they can compute $c_2 h^{-r} \bmod p$, and that will reveal the message m.

Problem 6 ***Block Cipher Leakage*** **(16 points)**

A hospital keeps a record, for each patient, of the patient's diseases. It is stored as a list of diseases along with a boolean indicating whether the patient has that disease or not:

```
acatamathesia: 0;ear infection: 0;heart disease: 1;...;xerophthalmia: 1;
```

Each record is encrypted. Assume that each "disease name: 0;" is exactly 16 bytes long (one block), disease names are all unique, and the list and order of diseases is public and the same for all patients.

A passive eavesdropper Eve intercepts two ciphertexts corresponding to the encryptions of Alice's and Bob's records. Assume that Eve has no prior knowledge of the disease status of any of the hospital's patients. The hospital uses the same key and **same IV** for encrypting each record.

(a) If the hospital uses AES-CBC mode with the same IV for every record, which of the following are true?

☐ Mallory can learn every disease for which Alice's boolean is equal to Bob's boolean

☐ Mallory can learn every disease for which Alice's boolean is not equal to Bob's boolean

☐ Mallory can always learn one disease for which Alice's boolean is equal to Bob's boolean, if any such disease exists

■ Mallory can always learn one disease for which Alice's boolean is not equal to Bob's boolean, if any such disease exists

☐ Mallory can never learn two diseases for which Alice's boolean is equal to Bob's boolean

■ Mallory can never learn two diseases for which Alice's boolean is not equal to Bob's boolean

■ Mallory can learn whether Alice and Bob have the same boolean for all diseases

☐ Mallory cannot learn anything about Alice and Bob's booleans

Solution: Because of the nature of CBC, the ciphertexts will be exactly the same until the first difference – at which point all the subsequent ciphertexts will be different. So Mallory learns a variable number of diseases where Alice and Bob's booleans are identical, and exactly one disease where their booleans are different (the first disease with a different boolean).

Some people told us that they interpreted "can" in the first two options as "can sometimes" (i.e., there exist situations where Mallory can). Since we put "can always" in several other options but not in the first two (our mistake), we thought this was a reasonable interpretation. So, we decided to also award credit if you selected both of the first two options, based on a "can sometimes" interpretation for both (but not if you selected just one of them). For the first two options, we are not awarding partial credit for getting just one of them correct.

Some people told us that they interpreted the next-to-last option as "for each disease, Mallory can learn whether Alice and Bob have the same boolean for that disease." We had been intending this as "Mallory can learn whether (for all diseases, Alice and Bob have the same boolean)", but in retrospect, this was ambiguous. We decided to accept both interpretations and grade accordingly.

(b) If the hospital uses AES-CTR mode with the same IV for every record, which are true?

- ■ Mallory can learn every disease for which Alice's boolean is equal to Bob's boolean
- ■ Mallory can learn every disease for which Alice's boolean is not equal to Bob's boolean
- ■ Mallory can always learn one disease for which Alice's boolean is equal to Bob's boolean, if any such disease exists
- ■ Mallory can always learn one disease for which Alice's boolean is not equal to Bob's boolean, if any such disease exists
- ☐ Mallory can never learn two diseases for which Alice's boolean is equal to Bob's boolean
- ☐ Mallory can never learn two diseases for which Alice's boolean is not equal to Bob's boolean
- ■ Mallory can learn whether Alice and Bob have the same boolean for all diseases
- ☐ Mallory cannot learn anything about Alice and Bob's booleans

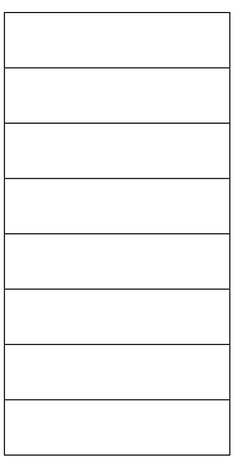
Solution: Since CTR doesn't have the same cascading effect as CBC, Mallory can tell for each disease whether Alice's boolean and Bob's boolean are the same.

Problem 7 *Memory safety exploits* **(26 points)**

The following code allows you to print characters of your choice from a string. It runs on a 32-bit x86 system with **stack canaries enabled**, but no other memory defense methods in use. Assume local variables are pushed onto the stack in the order that they are declared, and there is no extra padding, saved registers, or exception handlers. (These are the same assumptions as in homework 1.) Note that `scanf("%d", &offset)` reads a number from the input, converts it to an integer, and stores it in the `offset` variable.

```
1  void foo() {
2    char buf[300];
3    gets(buf);
4  }
5
6  int main() {
7    char *ptr;
8    int offset = 0;
9    char important[12] = "sEcuRitY!!!";
10   while (offset >= 0) {
11     scanf("%d", &offset);
12     ptr = important + offset;
13     printf("%c\n", *ptr);
14   }
15   foo();
16   return 0;
17 }
```

(a) Draw the stack, when at the point in time when line 12 of the code is executing, by filling in the diagram below. Label the location of sfp, rip (saved return address), stack canary, and the `ptr`, `offset`, and `important` variables, for `main`'s stack frame. Each empty box represents 4 bytes of stack memory. If a value spans multiple boxes, label all of them.



Solution:

rip
sfp
canary
ptr
offset
important
important
important

(b) Peyrin informs you that this code contains a vulnerability which leaks the value of `main`'s stack canary. Which sequence of inputs would leak this information? Fill in the blanks below.

_______ \n _______ \n _______ \n _______ \n

Solution: Since bounds aren't checked, use `ptr` to read off the stack canary: 20\n 21\n 22\n 23\n

(c) Next, suppose you want to develop a reliable arbitrary-code-execution exploit that works by overwriting `foo`'s entire return address, so that when `foo` returns, your shellcode will be executed. You first supply the string from part (b) to learn the value of the stack canary, followed by the string '-1\n', followed by a carefully chosen third string of some length. Write the *minimum* possible length of the third string, to achieve this. Assume your shellcode is 100 bytes long and it cannot be shortened.

Solution: 312 bytes. The stack frame for `foo` looks like

rip
sfp
canary
buf
⋮
buf

We're going to overflow `buf`, so we need 300 bytes for `buf`, plus 4 bytes for `foo`'s canary, plus 4 bytes for `foo`'s sfp, plus 4 bytes for overwriting `foo`'s rip. We can store the shellcode within `buf`, so we don't need another 100 bytes for it. Notice that the canary is the same for every function, so after learning `main`'s canary, we know that the same value will be used for `foo` as well.

We'll also accept 313, in case you thought that you need a newline at the end (`gets` doesn't actually require a newline—you could actually omit the newline and substitute it with end of file—but that's beyond the scope of what we're testing in this class).

(d) Your friend claims that it's not necessary to overwrite the entire return address to achieve arbitrary code execution: if you don't get unlucky with where certain addresses happen to fall, it's possible to reduce the length of the third string in part (c) to 304 bytes or 305 bytes, using an exploit that overwrites the least significant byte of sfp. Is she right?

● Yes ○ No

Solution: This is like Project 1, Question 4.

Suppose we use a 304-byte string, that doesn't contain any null bytes or newline characters. `gets()` will append a null byte, so it will write 304 bytes plus a null byte. This overwrites all of `buf`, overwrites `foo`'s canary, and then overwrites the least significant byte of `foo`'s sfp with a null byte. Because the least significant byte of `foo`'s sfp has been replaced with `0x00`, its value is now somewhat smaller (it now points somewhere lower in the stack), and it is likely the sfp will now be pointing to somewhere in the middle of `buf`. After `foo` returns, its sfp will be restored into `%ebp`. Now when `main`'s epilogue executes, it will store this value into `%esp`, then pop 4 bytes from there, and then return, i.e., pop a 4-byte value and transfer control there. We can anticipate where `foo`'s sfp was pointing, i.e., where in `buf` these 8 bytes are located, and we can make sure that the second 4 bytes contain the address of our shellcode. This will work as long as the original value of `foo`'s sfp doesn't end in `0x00`–`0x37`, since then replacing it with `0x00` will decrease it by at least `0x38` bytes, which is enough that it points into somewhere in `buf` with at least 8 bytes available for storing our bogus sfp and rip. Phew. That was pretty complicated.

If you choose a string that ends in a newline, you'll need 305 bytes, as `gets` replaces the newline with a null byte.

(e) The developers propose to fix the program by replacing lines 12–13 with the following code. Fill in the blank inside the if-statement to make the fix correct.

```
12    if (_____________________________________) {
13      ptr = important + offset;
14      printf("%c\n", *ptr);
15    }
```

Solution: `0 <= offset && offset < 12` or `0 <= offset && offset < sizeof(important)`

Unfortunately the fix isn't available yet. Unsettled by your exploit, the sysadmins **enable ASLR** for the stack and the heap as a temporary defense for the rest of this question.

You discover that the code (text segment) is not randomized, and you learn the address of a `ret` instruction. For the purpose of this question, you can assume that `ret` is a one-word instruction which is equivalent to `pop %eip`. In other words, it loads the instruction at `$esp` into the `$eip` and increments `$esp` by one word.

(f) Which exploit technique would be appropriate for an arbitrary code execution exploit against this code, given this new information?

● ROP

○ Overwrite the first byte of sfp

○ TOCTTOU

○ Exploit a format string vulnerability

Solution: You'll need to use ROP.

I don't know of any easy way to modify the exploit in part (d) to work in this setting (e.g., we don't know the address of our shellcode, so we can't put a bogus rip pointing to our shellcode in buf and hope the modified sfp will point there).

(g) Provide bounds on x, such that the input 'x\n' will cause `ptr` to point somewhere in the region where `buf` will appear.

____________ ≤ x ≤ ____________

Solution: -312 ≤ x ≤ -13.

Since the offset is signed we can input a negative number to move down the stack to `foo`'s frame. The stack will look like this (with the stack frame for `main` on top, and the stack frame for `foo` below it):

rip
sfp
canary
ptr
offset
important
important
important
rip
sfp
canary
buf
⋮
buf

We count from the start of `important` to the first byte of `buf`: len(`rip`) + len(`sfp`) + len(`canary`) + len(`buf`) = 4 + 4 + 4 + 300 = 312 bytes. So we have -312 ≤ x ≤ -13.

(h) Your exploit constructs an input as follows: first supply the string from part (b) to learn the value of the stack canary, followed by the string 'x\n' (with x chosen somehow based on part (g)) to set `ptr` appropriately, followed by a carefully chosen third string that is composed from multiple pieces. Below, select all possibilities for how to choose the third string so that the shellcode will be executed with probability at least 1/2.

Assume SHELLCODE is a 100-byte string containing the shellcode you want to execute, CANARY is the 4-byte value of the canary (learned using the technique from part (a)), `gadget` is the 4-byte address of the `ret` instruction you found, and NOPSLED is a 200-byte string containing many

NOP instructions. Beware that `gets` will replace the newline at the end of your third string with a null byte, so your exploit might need to deal with this.

1. First 300 bytes of the third string:

- ☐ SHELLCODE * 3
- ☐ SHELLCODE + 'a' * 196 + CANARY
- ■ NOPSLED + SHELLCODE
- ☐ `gadget` * 75

Solution: Since the null byte from `gets` will overwrite the last byte of `ptr` (as explained below), we need a NOP sled to give us a higher probability of reaching the shellcode.

2. Next 12 bytes of the third string:

- ○ `gadget` * 3
- ○ `gadget` * 2 + CANARY
- ○ CANARY * 3
- ○ CANARY + 'a' * 4 + CANARY
- ● CANARY + `gadget`*2
- ○ CANARY * 2 + 'a' * 4

Solution: We need to overwrite the canary correctly, we don't really care about the `ebp`, and we want to overwrite the `rip` with the address of the `ret` instruction to begin execution of our ROP chain (see below for a detailed explanation of the exploit).

3. Next bytes of the third string: (fill in the blank with a Python expression; your expression may reference SHELLCODE, NOP, CANARY, `gadget`, and 'a's, though you won't need them all)

Solution: `gadget` * 4

TL;DR: recursively chaining `gadget` will keep popping us up the stack until we reach `ptr` and jump to our shellcode!

When putting together all the pieces, this exploit string overwrites all of buf with a NOP sled and the shellcode, overwrites `foo`'s canary with the correct value, overwrites `foo`'s rip with `gadget`, and overwrites the next four 4-byte words in the stack (all of `important` and `offset`) with `gadget` as well, and finally overwrites the least significant byte of `ptr` with a null byte. When `foo` returns, it will add 4 to `%esp` and transfer control to the address `gadget`. At `gadget` there is a return instruction, so the CPU will execute another return instruction, which will add 4 to `%esp` and transfer control to the next address on the stack—which also happens to be `gadget`.

This continues for a while, until eventually we get to `ptr` and transfer control to the address stored in `ptr` (which, remember, had its least significant byte overwritten). Since `gets` appends a null byte, the least significant byte of `ptr` has been overwritten with a null byte, causing `ptr` to point to an address lower on the stack than its value before the overflow. In particular, there is a good chance that it will be pointing into the middle of `buf`, so when the last return instruction transfer control there, we'll be transferring control into somewhere in the middle of `buf`. The exact place in `buf` is random (it depends on the least significant byte of the original value of `ptr`, which is randomized by ASLR). However, since `buf` is big enough (300 > `0xff`), we have a high probability of landing somewhere in the NOP-sled and sliding to our shellcode.

This also explains why we need a NOP sled in part 1, since without the NOP sled we'd land somewhere random in the middle of `buf` and it's unlikely we'd hit exactly the start of the shellcode.

Note that since `ret` exists in pretty much any given program, this ROP attack may be very possible if you have an overflow and a mutable pointer.

4. Final byte of the third string:
\n

Selected C Manual Pages

```
char *gets(char *s);

       gets()  reads  a  line from stdin into the buffer pointed to by
       s until either a terminating newline or EOF, which it replaces
       with a null byte ('\0').

int printf(const char *format, ...);

       The  functions  in the printf() family produce output according to
       a format.  The functions printf() and vprintf() write output to
       stdout, the standard output stream.

       The format specifier %c prints a single character: the argument
       is interpreted as a character and printed.

int scanf(const char *format, ...);

       The scanf() family of functions scans input according to format
       as described  below.  This format may contain conversion
       specifications; the results from such conversions, if any, are
       stored in the locations pointed to by the pointer arguments that
       follow format.

       The format specifier %d reads an integer, represented in decimal
       notation, and writes it to the location pointed to by the argument.
```