Nick Weaver
Fall 2018

# CS 161
# Computer Security

# Midterm 1

Midterm solutions updated October 2020 by CS161 FA20 course staff.

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.*

SIGN your name: ______________________

PRINT your class account login: `cs161-`______________________ and SID: ______________________

Name of the person sitting to your left: ______________________

Name of the person sitting to your right: ______________________

You may consult one sheet of paper of notes. You may not consult other notes, textbooks, etc. Calculators, computers, and other electronic devices are not permitted. We use Gradescope for grading so please write your answers in the space provided.

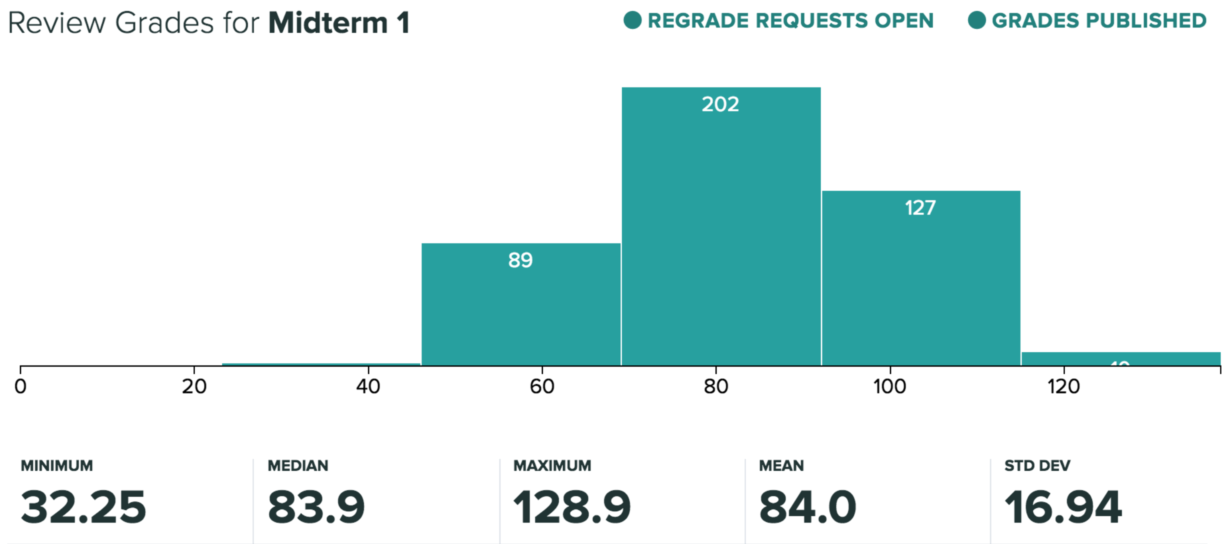
If you think a question is ambiguous, please come up to the front of the exam room to the staff. If we agree that the question is ambiguous we will add clarifying assumptions to the central document projected in the exam rooms.

You have 110 minutes. There are 11 questions, of varying credit (134 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

Some of the test may include interesting technical asides as footnotes. You are not responsible for reading the footnotes.

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

Grade distribution (out of 134 points):



**Problem 1** ***Cryptography True/False*** **(18 points)**

Answer the following cryptography questions true or false.

(a) Let $E_k$ be a secure block cipher. TRUE or FALSE: It is impossible to find two messages $m$ and $m'$ such that $m \neq m'$ and $E_k(m) = E_k(m')$, even if the attacker knows $k$.

● TRUE ○ FALSE

**Solution:** True. A block cipher needs to be a one-to-one function so it can be decrypted. If there existed $m \neq m'$ such that $E_k(m) = E_k(m')$, there would be no way to uniquely decrypt $E_k(m)$.

(b) Let $E_k$ be a secure block cipher. TRUE or FALSE: It is computationally difficult to find two pairs $(m, k)$ and $(m', k')$ such that $m \neq m'$, $k \neq k'$ and $E_k(m) = E_{k'}(m')$.

○ TRUE ● FALSE

**Solution:** False. Let $k' \neq k$ and $m' = D_{k'}(E_k(m))$. With high probability we have $m' \neq m$ as desired, and $E_{k'}(m') = E_{k'}(D_{k'}(E_k(m))) = E_k(m)$.

(c) Let $\text{MAC}_k$ be a secure MAC. TRUE or FALSE: It is computationally difficult to find messages $m$ and $m'$ such that $m \neq m'$ and $\text{MAC}_k(m) = \text{MAC}_k(m')$, even if the attacker knows $k$.

○ TRUE ● FALSE

**Solution:** False. MACs don't make any guarantees about whether two different values might have the same MAC.

It depends on the MAC: In particular, AES-MAC is secure but does not have this property. You can have multiple messages all with the same MAC, because you just take the intermediate values: you mac a message M, and then ”roll back” a single block. HMAC does have this property however. And this is why HMAC-accept-no-substitutes!

(d) Let $H$ be a cryptographic hash function. TRUE or FALSE: $H(M)$ provides confidentiality for the message $M$.

○ TRUE ● FALSE

**Solution:** False. Hashes are deterministic, so an attacker could tell if the same message was sent twice. Also, an attacker could test a guess at $M$.

(e) HMAC-DRBG does not have rollback resistance.

○ TRUE ● FALSE

**Solution:** False. Intuitively, the underlying hashes of the HMAC make it hard to revert to a previous state, since a cryptographic hash is one-way.

(f) Diffie/Hellman is secure in the presence of an active adversary.

○ True ● False

**Solution:** False. A man-in-the-middle can intercept Alice's $g^a$ and send $g^m$ to Bob, and intercept Bob's $g^b$ and send $g^m$ to Alice. Then Alice thinks the shared key is $g^{am}$ and Bob thinks the shared key is $g^{bm}$, and since the adversary knows $g^a$, $g^b$, and $m$, the adversary knows both secrets.

(g) Properly constructed RSA Signatures provide both integrity and authenticity.

● True ○ False

**Solution:** True. An attacker can't generate a valid signature without knowing the secret key, so the attacker can't modify the message without being detected (integrity), and the attacker can't forge a message with a valid signature (authenticity).

(h) El Gamal encryption provides confidentiality but it does not provide integrity or authentication.

● True ○ False

**Solution:** True. El Gamal provides only confidentiality. For example, you can replace ciphertext $(c_1, c_2)$ with $(c_1, 2c_2)$, and the recipient will think the message is $2m$ instead of $m$.

(i) In examining a certificate we need to consider how we obtained the certificate as well as the certificate's contents and signatures.

○ True ● False

**Solution:** Certificates are signed, so we don't care where the certificate was obtained, as long as the signature is valid.

**Problem 2** ***Potpourri*** **(18 points)**

(a) Instead of storing user input on the stack, you decide to create a new section of memory (separate from code, static, heap, and stack) for storing user input. You also put a 64-bit canary at the top (largest memory address) of the section. Name one memory-safety vulnerability that this prevents.

> **Solution:** This prevents a simple buffer overflow attack from changing return addresses on the stack.

(b) Name one memory-safety issue that the scheme from part (a) fails to prevent.

> **Solution:** This does not prevent buffer overflows from overwriting other user input stored in the section. Format string vulnerabilities can still allow the attacker to read arbitrary parts of memory. Programmer sloppiness is also a possibility as copying user input into a local variable stored in the stack (through strcpy etc.) can still cause buffer overflows to overwrite the return address.

(c) TRUE or FALSE: In a threat detection systems, false negatives can be catastrophic, but false positives are always harmless.

○ TRUE ● FALSE

> **Solution:** False, false positives can take time, money, and other resources to address. False positives can make a good detector/alarm unusable even if it has a very low false negative rate.

(d) Which of the following are recommended ways to protect a password database? (Select all that apply.)

■ Salting Passwords □ Using a Fast Hashing Function

□ Encrypting Passwords ■ Using a Slow Hashing Function

> **Solution:** Salting passwords prevents a dictionary attack, since the attacker needs to perform one dictionary attack per user instead of one dictionary attack for the entire database.
>
> Encryptng passwords isn't recommended because if you store the key with the encrypted passwords and you get hacked, then all the passwords are immediately broken.
>
> Using a fast hashing function isn't recommended because it allows the attacker to perform a dictionary attack faster. A slow hashing function is better because it makes the dictionary attack slower.

(e) A heap overflow or use-after-free vulnerability can allow the attacker to overwrite the `vtable` pointer of an object (that is, the pointer at the start of a C++ object that points to the actual methods for the function, basically a pointer to an array of function pointers). Can this bypass stack canaries without additional information?

● Yes ○ No

> **Solution:** There is no stack canary before the `vtable` pointer, so an attacker can overwrite the pointer without modifying a stack canary.

(f) At what rank did Grace Hopper retire?

- ○ Lieutenant Colonel
- ○ Captain
- ● Rear Admiral
- ○ Brigadier General

**Solution:** This was an attendance question for a group of students who were attending the Grace Murray Hopper conference, and Nick wanted to make sure that they knew that Admiral Grace Hopper was an Admiral.

(g) Alice generates a MAC on her homework answers that she stores with her homework answers in a secret remote server. When she needs to submit her homework, she uses the MAC to check that her answers have not been tampered with. Only she has the key needed to generate the MAC. Which of the following apply in this scenario?

- ○ Integrity and Confidentiality
- ○ Authentication and Confidentiality
- ● Integrity and Authentication
- ○ Only Integrity

**Solution:** MACs provide integrity and authentication. No one else has the key, so an attacker can't tamper with Alice's answers (integrity) or forge answers with a valid MAC (authenticity).

(h) Which of the following attacks can be used against a crypto system? (Select all that apply.)

- ■ Side-Channel
- ■ Chosen-ciphertext
- □ Rolling-regression
- ■ Rubber-Hose Cryptanalysis
- ■ Chosen-plaintext

**Solution:** Side-channel attacks take advantage of faulty implementations that leak information (e.g. a correct password validates faster than an incorrect password).

Rolling regression is unrelated to cryptography.

Chosen-plaintext and chosen-ciphertext are specific classes of attacks where an attacker finds a way to encrypt and decrypt arbitrary messages, respectively.

Rubber-hose cryptanalysis tries to directly get the secret from the person in real life, through blackmail or coercion.

(i) "Crypto" means:

- ■ Cryptography
- □ Kryptonite
- □ Cryptocurrency
- □ CryptoKitties

**Solution:** Attendance question. Crypto stands for cryptography.

(j) The Magic Word is:

☐ Adava Kedavra

☐ Windgardium Leviosa

■ Stupify

☐ Crucio

**Solution:** Attendance question.

**Problem 3** ***Security Principles*** **(12 points)**

Write the best match for which security principle each situation.

Four CS 161 students, Chiyo, Habiba, Mr. Anderson, and Not Outis, decided that after learning about security principles and buffer overflows, they could implement their own distributed database (a database across multiple machines) with a focus on security!

(a) Mr. Anderson suggests code their database in a higher-level programming language since they could avoid common security problems later on. Which security principle did he to use here?

> **Solution:** Design in Security from the Start

(b) Let's say they start coding their database and realized that a malicious user on one machine could corrupt their database. As a result, Habiba wants permission from at least 50% database users before a machine can be taken down. Which security principle is she using here?

> **Solution:** Division of Trust

(c) The database the students built was password-protected for modification and they use a snippet (like the following) everywhere to check passwords:

```
String password = getPassword("user");
if (!password.equals(enteredPassword)) error();
```

Not Outis eventually forgets to put this snippet to check the passwords. What security principle does this violate?

> **Solution:** Ensure Complete Mediation

(d) To encrypt the data, Not Outis decides to take each piece of data and rotate the bytes in it by a fixed amount. It figured that since their database was closed source, no one would figure out how they were encrypting things. What security principle does this violate?

> **Solution:** Shannon's Maxim or Kerckhoff's Principle

(e) Mr. Anderson decides that new users should automatically get privileged access in order to set up their account to access whatever items they needed. After 1 hour, they would be dropped back to regular permissions, an administrator would be notified of changes, and they could revert changes if necessary. What security principle says this is not a good idea?

> **Solution:** Fail-Safe Defaults

(f) After fixing all previous problems, Chiyo decides to refactor their encryption code into its own module since a lot of it was spread across multiple modules. She also put all non-encryption code in a sandbox so that no vulnerabilities in those modules could effect the overall security of the database. What security principle is she trying to follow? What is she trying to minimize the size of?

> **Solution:** Privilege Separation, TCB

## Problem 4 *Go With The Control Flow* (14 points)

The code below runs on a 32-bit Intel architecture. No defenses against buffer overflows are enabled. The code was not compiled to produce a position independent executable. No optimizations are enabled, and the compiler does not insert padding or reorder stack variables, which means `buffer` is at a lower address than `fp`.

```
1  int run_command(char *cmd) {
2      return system(cmd);
3  }
4  int print_hello(char *msg) {
5      printf("Hello %s!\n", msg);
6      return 0;
7  }
8  int main() {
9      int (*fp)(char *) = &print_hello;
10     char buffer[8];
11     gets(buffer);
12     fp(buffer);
13 }
```

Note that the syntax `int (*fp)(char *)` indicates that `fp` is a pointer to a function which takes in a `char *` and returns an `int`.

(a) What line contains a memory vulnerability? What is this vulnerability called?

> **Solution:** Line 11. Buffer overflow!

(b) At line 12, we have that `%ebp = 0xbfdead20` and `&print_hello = 0x08cafe13`. Fill in the Python egg below to give an input which will **overwrite the return address of `main`**, causing the execution of the shellcode after the program returns from `main`.

```
print 'A' * _____ + '_________________' + 'AAAA' + '_________________' + SHELLCODE
```

> **Solution:** `print 'A' * 8 + '\x13\xfe\xca\x08AAAA\x28\xad\xde\xbf' + SHELLCODE`
>
> First, write 8 bytes to overflow the buffer.
>
> Line 12 calls the function `fp` before the function returns, so if we overwrite `fp` with garbage, the program will try to dereference the garbage as the address of a function and crash. So we need to overwrite the function pointer `fp` with its original value `0x08cafe13`.
>
> Next, write 4 bytes of garbage to overwrite the sfp.
>
> Finally, overwrite the rip with the address of shellcode, which is 4 bytes above the rip: `0xbfdead20 + 4 = 0xbfdead24`.

(c) Which of the following would sometimes or always prevent the code that you gave in part (b) from working? (Select all that apply.)

- ■ ASLR (same as part 5 on the project)
- ■ Selfrando
- ■ W^X
- ■ Using a memory-safe language instead of C

> **Solution:** ASLR and Selfrando (ASLR that also randomizes function locations) stop the exploit because you no longer have an absolute address to overwrite the rip with.
>
> WˆX stops the exploit because you can't execute the shellcode that you wrote on the stack (since the stack is writable, and thus not executable).
>
> Using a memory-safe language always stops buffer overflow attacks.

(d) "I know," says Louis Reasoner, "let's add stack canaries to make this impossible to exploit!" Obviously this doesn't work. Fill in the Python egg below to give an input which will cause the execution of `run_command("/bin/sh")`. At line 12, we have that `%ebp = 0xbfdead20` and `&run_command = 0x08c0de42`. HINT: Note that `gets` can read in a `NUL` byte (`\x00`), even in the middle of its input.

`print '____________________________________________________________'`

> **Solution:** First, write the null-terminated string `/bin/sh` into buffer.
>
> Next, overflow `fp` so it's pointing at the address of the `run_command` function. This causes line 12 to call `run_command` with the argument in `buffer`, which is `/bin/sh`.
>
> Note that this exploit never overwrites the stack canary, which would be above the `fp` local variable.
>
> `print '/bin/sh\x00\x42\xde\xc0\x08'`

(e) Which of the following would sometimes or always prevent the code that you gave in part (d) from working? (Select all that apply.)

- ☐ ASLR (same as part 5 on the project)
- ■ Selfrando
- ☐ WˆX
- ■ Using a memory-safe language instead of C

> **Solution:** Project 1-style ASLR doesn't stop the exploit because it doesn't randomize the code section, and the address of `run_command` is located in the code section and thus stays the same every time the program runs.
>
> Selfrando (ASLR that also randomizes function locations) stops the exploit because it would change the address of `run_command` every time the program is run, so you wouldn't know its absolute address.
>
> WˆX doesn't stop the exploit because the exploit never tries to execute code on the stack. (`/bin/sh` is just a string argument that gets passed to `run_command`, not actual x86 instructions.)
>
> Using a memory-safe language always stops buffer overflow attacks.

**Problem 5** ***Ben Bitdiddle's Preconditions*** **(8 points)**

Ben Bitdiddle did not do a good job at coming up with a set of preconditions for some functions. For each code block, explain why with a short example the given preconditions are **not** sufficient to ensure memory safety by giving a small example.

(a)

```
/*  array_of_strings != NULL
    n <= size(array_of_strings)
    max_size > 0
    for all i . 0 <= i < n ==>
         array_of_strings[i] != NULL and is a NUL-terminated string */
char *
concat_all(char *array_of_strings[],  size_t n, size_t max_size) {
    char *concat = calloc(max_size, sizeof(char));
    if (!concat) return NULL;
    size_t space_used = 0;
    for (size_t i = 0; i < n; i++) {
        char *s = array_of_strings[i];
        size_t len = strlen(s);
        strncpy(concat + space_used, s, max_size - space_used - 1);
        space_used += len;
    }
    return concat;
}
```

Explanation:

**Solution:** Consider `concat_all({"abcde", "fghi"}, 2, 5)`. After the first loop iteration, we will have `space_used = 5, max_size = 5`, and `concat = "abcd\0"`. Then because of integer overflow, we have `max_size - space_used - 1 = (size_t) -1` (which is really big!) On the next iteration we write out-of-bounds, and this is a heap buffer overflow.

(b)

```
/*  arr != NULL
    n <= size(arr)
    for all i . 0 <= i < n ==> 0 <= arr[i] < n */
int solve_interview_question(int *arr, size_t n) {
    for (size_t i = 0; i < n; i++)
        arr[arr[i]] *= -1;
    for (size_t i = 0; i < n; i++)
        if (arr[i] < 0)
            return i;
    return 0;
}
```

Explanation:

**Solution:**

Consider `arr = {1, 1}`. Then all of the preconditions are met, but this accesses `arr[-1]` (in the second loop iteration) which may not be defined.

**Problem 6** ***Greetings Professor Falken!*** **(9 points)**

Consider the code below.

```
1 void launch_nuclear_missiles() {
2     puts("Launching the nukes...");
3     /* code to launch nuclear missiles here */
4     exit(1);
5 }
6
7 #define MAX_INPUT 8
8 int main() {
9     char *correct_password = malloc(MAX_INPUT * sizeof(char));
10     strcpy(correct_password, "S3creT\n");
11     while (!feof(stdin)) {
12         char *user_password = malloc(MAX_INPUT * sizeof(char));
13         fgets(user_password, MAX_INPUT, stdin);
14         if (strcmp(user_password, correct_password) == 0)
15             launch_nuclear_missiles();
16         free(user_password);
17         free(correct_password);
18         puts("Wrong password, try again!");
19     }
20 }
```

All compiler optimizations are disabled, and both the source and binary are not available to David Lightman, who's trying to log in to play a game. Consider the following (buggy) interaction:

1. David inputs "Hello" followed by a newline.
2. The program outputs "Wrong password, try again!".
3. David inputs "Joshua" followed by a newline.
4. The program outputs "Launching the nukes...", and then the nukes are launched.[1]

(a) Which memory safety vulnerability is present in this code?

> **Solution:** Use after free. Line 17 frees the `correct_password` variable on the heap, but the next time the while loop runs, the variable gets used again at line 14.

(b) Explain why this issue leads to the behavior David observes.

> **Solution:** The memory for `correct_password` is reused by the next `malloc` for `user_password`. Therefore the second input is always correct as `correct_password == user_password`.

(c) How could you fix this issue in the code?

> **Solution:** Delete line 17.

[1] This immediately vaporizing millions of humans and wildlife on impact, beginning World War III and eventually wiping out most of the world due to an extended nuclear winter. This is why you don't hack into systems without permission. If you want to understand more how nuclear command, control, and decision making works, the two books to read are <u>Command and Control: Nuclear Weapons, the Damascus Accident, and the Illusion of Safety</u> by Eric Schlosser, and <u>The 2020 Commission Report on the North Korean Nuclear Attacks Against the United States (A Speculative Novel)</u> by Jeffrey Lewis.

**Problem 7** ***Fail Caesar*** **(12 points)**

A student at a well known Junior University decided to write their own Caesar Cipher after learning about them in their computer security class. Unfortunately for the student, they fell asleep during the lecture on memory safety. (Note: The `atoi()` function converts the initial portion of the string to an integer, returning 0 in case of an error.)

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  void encrypt(int offset, char plaintext[]) {
4      char ciphertext[64];
5      memset(ciphertext, 0, 64);
6      int i = 0;
7      fgets(plaintext, 64, stdin);
8      while (plaintext[i]){
9          ciphertext[i] = plaintext[i] + offset;
10         i++;
11     }
12     printf(ciphertext);
13 }
14
15 int main(int argc, char *argv[]){
16     char buffer[64];
17     int offset = 0;
18     if (argc > 1) offset = atoi(argv[1]) % 26;
19     while (!feof(stdin)){
20         memset(buffer, 0, 64);
21         encrypt(offset, buffer);
22     }
23     return 0;
24 }
```

(a) What line contains a memory vulnerability? What is this vulnerability called?

> **Solution:** The vulnerable code is on line 12. This is a format string vulnerability, since the attacker controls the first argument of `printf`.

(b) Give a file that, when input to the command `failcaesar` with no arguments, will cause the program to crash.

> **Solution:** Either a lot of `%s` or `%n` items, since these try to dereference pointers on the stack, which will likely point to undefined parts of memory.

(c) How would you change the line to fix the vulnerability?

> **Solution:** Change `printf(ciphertext)` to `printf("%s", ciphertext)` or `fputs(ciphertext, stdout)`. Also accept `puts(ciphertext)` or `printf("%s\n", ciphertext)` although they add a trailing newline.

(d) The student's friend who was awake for the memory safety lecture tells them to enable stack canaries to make their code more secure. If an attacker does not have time to perform a bruteforce attack, does enabling stack canaries prevent this code from being exploited? Explain why or why not.

**Solution:** No, in a format string vulnerability a malicious user can write directly to a desired address in memory without making consecutive writes up the stack. As such, Mallory can write around the stack canary to overwrite the return instruction pointer.

**Problem 8** ***A Lack of Integrity...*** **(9 points)**

Alice and Bob want to communicate. They have preshared a symmetric key $k$. In order to send a message $M$ to Bob, Alice encrypts it using AES-CBC, and sends the encryption to Bob. (You may assume that $M$'s length is divisible by the AES-CBC block length and that characters are 8 bits, so no padding is necessary.) Recall that the actual message sent is $IV||E(M)$, that is, the IV is prepended to the message and sent all as a single stream of bytes. Alice uses a random IV for each message.

In order to make sure that Bob is listening, they agree to using *pingback* messages. If Alice sends a message whose plaintext begins with the two bytes "PB", then Bob sends back the rest of the message *in plaintext*. For example, if Alice sends AES-CBC$_k$("PBI Love CS 161!"), then Bob responds "I Love CS 161!" without any encryption.

Alice uses the protocol to communicate some message $M$ to Bob. Assume $M$ is not a pingback message. Mallory, a man-in-the-middle attacker, decides to attempt to trick Bob into generating a pingback message. She thus sends the message $IV'||IV||E(M)$, where $IV'$ is a random 128b string.

(a) With what probability will Mallory's message trigger a pingback message?

> **Solution:** 1 in $2^{16}$. The first 2 characters = 2 bytes = 16 bits of plaintext need to be exactly "PB". The block cipher decryption of Mallory's message will be effectively random, so the probability the first 16 bits are exactly "PB" is $\frac{1}{2^{16}}$.

(b) If Mallory's message triggers a pingback message, what does Mallory receive?

> **Solution:** Recall block cipher decryption: $P_i = D_k(C_i) \oplus C_{i-1}$. (This question could also be solved by looking at the decryption diagram.)
>
> Consider Mallory's message $IV'||IV||E(M)$. The IV is $IV'$, the first block of ciphertext is $IV$, and the subsequent blocks of ciphertext are $E(M)$.
>
> The first block of ciphertext $IV$ is random bytes, so the result of passing it through block cipher decryption is 128 random bits = 16 random bytes. The decryption is XOR'd with the IV, which is the random string $IV'$, so the resulting plaintext block is 16 random bytes. If the message triggers a pingback, the first two characters must be "PB", and those are not sent in the pingback. So the first thing Mallory receives is 14 random bytes.
>
> The rest of the ciphertext is $E(M)$. At the second block (where the first block of $E(M)$ is), the previous block of ciphertext is the first block of ciphertext, which happens to be $IV$. This is exactly equivalent to decrypting Alice's real message $IV||E(M)$ in CBC mode. So the next thing Mallory receives is the real message.

(c) How can Alice and Bob change their protocol to prevent this attack?

> **Solution:** Use a MAC, so Mallory can't tamper with the message.

**Problem 9** ***Screwups in Inserting an IV*** **(15 points)**

Alice encrypts two messages, $M_1$ and $M_2$ using the same IV/nonce and a deterministic padding scheme (when appropriate for the particular mode) using AES (a 128b block cipher). Eve, the Eavesdropper, knows the plaintext of $M_1$, that each block of $M_1$ is different, that $M_1$ is 120 *bytes*, and that Alice never sends any bytes she doesn't have to. Unbeknownst to Eve, it turns out that the messages differ only in the 21st byte of the two messages but are otherwise identical.

Yes, Alice screwed up. But how badly? For each possibility, select *all* which apply.

(a) If Alice used AES-ECB (Electronic Code Book), Eve is able to determine which of the following about $M_2$:

- ■ That $M_2$ is exactly 120B long
- ☐ The entire plaintext for $M_2$
- ■ The entire plaintext for $M_2$ except for the 2nd block
- ☐ That $M_2$ is less than 129B long but not the exact length
- ☐ The plaintext for only the first two blocks of $M_2$
- ☐ The plaintext for only the first block of $M_2$

> **Solution:** Exact length is known because the last ciphertext blocks for two messages must be identical: Attacker can deduce since the first message is 120B, and the second message has the same last block, the second message must be 120B.
>
> ECB is deterministic, so Eve knows every block of $M_2$ except the second block is identical to $M_1$. But she has no way of decrypting or learning anything about the second block, since she doesn't know the key for the block cipher decryption, and as a result, the output of the second block looks effectively random to her.

(b) If Alice used AES-CTR (Counter), Eve is able to determine which of the following about $M_2$:

- ■ That $M_2$ is exactly 120B long
- ■ The entire plaintext for $M_2$
- ☐ The entire plaintext for $M_2$ except for the 2nd block
- ☐ That $M_2$ is less than 129B long but not the exact length
- ☐ The plaintext for only the first two blocks of $M_2$
- ☐ The plaintext for only the first block of $M_2$

> **Solution:** Exact length is known because CTR mode naturally leaks the exact length of a message (it doesn't use padding).
>
> CTR with nonce reuse is essentially a one-time pad with pad reuse, since the plaintext is bitwise XOR'd with the output of the block cipher encryptions (the pad) to get ciphertext, and the ciphertext is XOR'd with the output of the block cipher encryptions (the pad) to get the plaintext.
>
> Eve can notice that all the bits in the encryptions are the same except possibly some in the 21st byte, deduce that a different bit of ciphertext corresponds to a different bit in the plaintext, and learn the entire plaintext for $M_2$.
>
> Alternatively, she could XOR the ciphertext $E(M_1)$ with $M_1$ to learn the pad, and then XOR the pad with the ciphertext $E(M_2)$ to learn $M_2$.

(c) If Alice used AES-CBC (Cipher Block Chaining), Eve is able to determine which of the following about $M_2$:

- ☐ That $M_2$ is exactly 120B long
- ☐ The entire plaintext for $M_2$
- ☐ The entire plaintext for $M_2$ except for the 2nd block
- ■ That $M_2$ is less than 129B long but not the exact length
- ☐ The plaintext for only the first two blocks of $M_2$
- ■ The plaintext for only the first block of $M_2$

**Solution:** The one bit difference in 2nd plaintext block completely changes the 2nd ciphertext block, which gets input to the 3rd AES encryption, which changes the 3rd ciphertext block...

The last ciphertext block is completely different, and effectively random to the attacker who doesn't know the key. The attacker cannot deduce the length by looking at the last cipher block.

Also, the attacker can only see that the first blocks of the two ciphertexts are identical, and deduce that the first block of $M_2$ is the same as the first block of $M_1$. Everything after the first block is effectively random to the attacker, so they can't deduce anything else.

(d) If Alice used AES-CFB (Ciphertext Feedback), Eve is able to determine which of the following about $M_2$:

- ■ That $M_2$ is exactly 120B long
- ☐ The entire plaintext for $M_2$
- ☐ The entire plaintext for $M_2$ except for the 2nd block
- ☐ That $M_2$ is less than 129B long but not the exact length
- ■ The plaintext for only the first two blocks of $M_2$
- ☐ The plaintext for only the first block of $M_2$

**Solution:** Exact length is known because CFB mode naturally leaks the exact length of a message (it doesn't use padding).

The attacker sees that the first blocks of the two ciphertexts are identical, so she deduces that the first block of $M_2$ is the same as the first block of $M_1$. Since the first blocks of the ciphertexts are identical, and CFB mode feeds the ciphertext into the block cipher encryption, the output of the second block cipher encryption is also identical. This output is then bitwise XOR'd with the second block of plaintext to get the second block of ciphertext. Eve can see that the two blocks of ciphertext differ in only one byte, and deduce that a different bit in the ciphertext for $M_2$ corresponds to a different bit in the plaintext for $M_2$.

The second block of ciphertext, which differs in one byte between the two messages, is passed into a block cipher encryption, which creates two completely different outputs for the two messages. Everything after this is completely different, so Eve can't learn anything more.

(e) If Alice did *not* screw up, which modes allow Eve to determine the exact length of a third message $M_3$ that is completely different from $M_1$ and $M_2$.

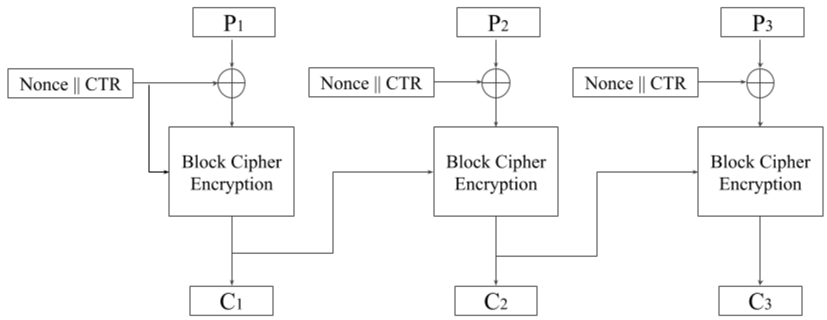
- ☐ AES-ECB
- ☐ AES-CBC
- ■ AES-CTR
- ■ AES-CFB

**Solution:** ECB and CBC use padding, so Eve can't learn the exact length of a message when the scheme is properly used.

CTR and CFB don't use padding, so they naturally leak the exact length of a message.

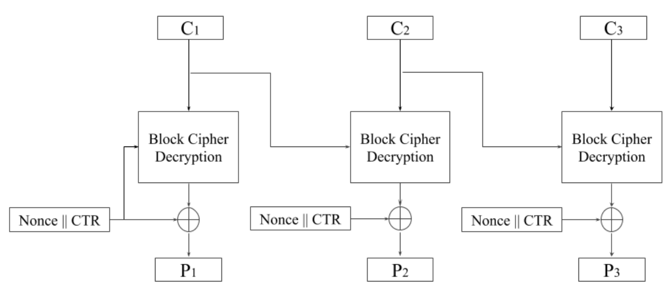
**Problem 10** ***No More Keys*** **(7 points)**

Frustrated by your newfound love of encryption schemes, your partner decides to throw away all of your secret keys. As a student in CS 161, you decide to make the best of a bad situation. You decide to design your own encryption scheme!




(a) Design the Decryption scheme.

**Solution:**




(b) This is IND-CPA:

○ TRUE ● FALSE

**Solution:** False. There's no secret key, so anyone can perform encryption or decryption.

(c) The encryption is parallelizeable:

○ TRUE ● FALSE

**Solution:** False. The output of each block cipher is used as an input to the next block cipher.

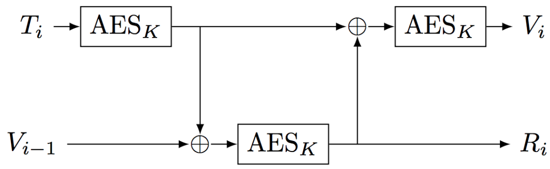
(d) The decryption is parallelizeable:

● TRUE ○ FALSE

**Solution:** True. Each block cipher decryption only requires ciphertexts as inputs, which are already known when decryption begins.

**Problem 11** ***Like Water off a DUHK's Back*** **(12 points)**

The ANSI X9.17/X9.31 is a fairly simple pRNG that was widely used based on a block cipher (commonly AES). The internal state $V$ and key $K$ are combined with the current time $T$ to update the state and produce a "random" value.




The current time is measured in microseconds as that is what the common operating system routines return. This is a strong pRNG as long as the initial state $V_0$ and the key $K$ are both high entropy and secret, and the block cipher is secure.

Unfortunately this scheme can fail badly when common mistakes are made. The standard never specified how to select $K$. So some implementations, rather than using a high-entropy source to seed a secret $K$, used a hardcoded key. The result is a catastrophic failure[2].

(a) If the attacker exactly knows $K$, $T_1$, and $R_1$, the attacker can then recover $V_0$. How?

> **Solution:** $R_1 = E_K(V_0 \oplus E_K(T_1))$
>
> Decrypt both sides:
>
> $D_K(R_1) = V_0 \oplus E_K(T_1)$
>
> XOR both sides with $E_K(T_1)$:
>
> $D_K(R_1) \oplus E_K(T_1) = V_0$

(b) Since one can then use this to calculate $R_0$ given $T_0$, what design principle for a good pRNG does this fail to implement?

> **Solution:** Rollback Resistance. Given the current state of the pRNG, you can calculate the previous state.

(c) If the attacker knows $T_0$ and $T_1$ with just millisecond resolution, the attacker can check to see if a possible candidate for $T_0$ and $T_1$ is consistent with guesses for $R_0$ and thereby know they found $V_0$. How many possible combinations of $T_0$ and $T_1$ may potentially need to be checked to determine $V_0$?

> **Solution:** There are 1,000 microseconds in a millisecond, so the attacker needs to try 1,000 possible times for $T_0$ and 1,000 possible times for $T_1$. This is (1,000)(1,000) = 1,000,000 combinations of $T_0$ and $T_1$.

[2]This was analyzed as the DUHK ("Don't Use Hardcoded Keys") attack, and it worked against FortiGate VPNs. For more details see https://duhkattack.com. This catastrophic failure mode is why it is no longer part of the standard suite of pRNGs.

# Foot-Shooting Prevention Agreement

I, ____________ , promise that once
Your Name

I see how simple AES really is, I will <u>not</u> implement it in production code even though it would be really fun.

This agreement shall be in effect until the undersigned creates a meaningful interpretive dance that compares and contrasts cache-based, timing, and other side channel attacks and their countermeasures.

X ______________________ ________

Signature Date