Weaver Fall 2017 — CS 161 Computer Security — Midterm 1

Midterm solutions updated February 2021 by CS161 SP21 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 6 questions, of varying credit (142 points total). The questions are of varying difficulty, so avoid spending too long on any one question. Use a #2/hb or softer pencil. For bubble questions, fill the bubble completely and clearly erase any mistakes.

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.

Problem 1 ***Multiple Guess*** **(36 points)**

(a) (4 points) You are writing some encryption routines. In it you reuse a nonce in a block cipher. Which are true? (Mark **all** which apply.)

- ● If using CTR mode, you lose all confidentiality against a known plaintext attack.
- ○ If using CTR mode, you lose all confidentiality against a chose ciphertext attack.
- ● Nick's spirit will reach out from your monitor and club you over the head for needlessly writing cryptographic code.
- ● If using CFB mode, you lose IND-CPA
- ○ If using CFB mode, you lose all confidentiality against a known plaintext attack.
- ● If using CBC mode, you lose IND-CPA
- ● If using ECB mode, you never had IND-CPA to lose

Solution: Recall that deterministic schemes are not IND-CPA secure. If the nonce/IV is reused, CFB and CBC mode both become deterministic, so you lose IND-CPA security.

ECB mode is deterministic regardless (there's no nonce/IV), so you never had IND-CPA security to lose.

Nick's spirit is well-known on this issue–as stressed in lecture repeatedly, you should not write your own cryptography.

Known plaintext attacks and chosen ciphertext attacks are out of scope as of spring 2021.

(b) (8 points) Mark **all** true statements:

- ● Stack canaries can not protect against all stack overflow attacks
- ● A format-string vulnerability can overwrite a saved return address even when stack canaries are enabled
- ● A one time pad is impractical because you can never reuse a one time pad
- ● ALSR, stack canaries, and NX all combined are insufficient to prevent exploitation of all stack overflow attacks
- ● RSA is only believed to be secure, there is no actual proof
- ● HMAC does not leak information about the message if the underlying hash is secure.
- ● Authentication implicitly also provides data integrity
- ● Salting a password does not prevent offline brute force attacks
- ● Failing to salt stored passwords ususally indicates programmer negligence

Solution: Left column, top to bottom:

True. Consider a stack overflow that only overwrites local variables. Stack canaries won't protect against this exploit.

True. Format string vulnerabilities can write to arbitrary locations in memory, essentially writing "around" the stack canary.

True. Recall that one-time pads lose all security if you reuse the key. This means that to send an n-bit message with one-time pad, you would need to first send an n-bit key...but if you've found a way to send an n-bit key securely, you could have just sent the message directly. Thus one-time pads are impractical.

True. Again, consider the overflow that only overwrites local variables. Stack canaries, ASLR, and NX combined won't protect against this exploit.

Right column, top to bottom:

True. There's no formal proof that RSA is secure. However, since no one in the security community has been able to break RSA in all the years since its introduction, we assume that it's secure.

True. Secure cryptographic hashes can't be reversed (given the hash output, it's hard to find what input produced that hash), so HMAC does not leak information about the message.

True. Authentication means you are sure that the message came from the original sender. This also implies that nobody has tampered with the message, i.e. the data has integrity.

True. Salting makes brute-force attacks harder, because the attacker has to run a separate brute-force attack for each user, but an attacker can still hash an entire dictionary of passwords with the salt attached (e.g. H(password123 ‖ salt), H(password1234 ‖ salt), etc.) and see which password produces a matching hash.

True. Storing passwords without a salt probably means someone has written their own crypto, which indicates programmer negligence.

(c) (4 points) You have multiple independent detectors in series so that if any detector triggers you will notice the intruder. Which are true? (Mark **all** which apply.)

- ● You are employing *defense in depth*.
- ● Your false negative rate will decrease.
- ● Your false positive rate will increase.
- ○ Your false positive rate will decrease.

Solution: This is a defense in depth approach because an attacker needs to defeat both detectors in order to carry out an attack.

If you alert when either detector alerts, then you expect to be alerting more often in general. Alerting more often causes fewer false negatives (failing to alert when there's an attack) but more false positives (alerting when there's no attack).

(d) (4 points) You have a non-executable stack and heap. Which are true? (Mark **all** which apply.)

- ○ An attacker can write code into memory to execute.
- ○ Buffer overflows are no longer exploitable
- ● An attacker can use Return Oriented Programming
- ● Format-String vulnerabilities may still be exploitable

Solution: Left column, top to bottom:

False. Any data the attacker writes into memory will be marked non-executable.

False. Consider overwriting local variables. Non-executable stack and heap doesn't prevent this exploit.

Right column, top to bottom:

True. Return-oriented programming (ROP) executes fragments of code (gadgets) that are already in memory.

True. Format-string vulnerabilities don't require executing any code on the stack or heap. For example, consider writing `%x%x%x%x` as input to leak memory on the stack.

(e) (4 points) Which are true about RSA encryption? (Mark **all** which apply.)

- ○ RSA encryption without padding is IND-CPA
- ○ RSA encryption provides integrity
- ○ Padding involves simply adding 0s
- ● RSA signatures provide integrity

Solution: Left column, top to bottom:

False. RSA encryption without padding is deterministic, and deterministic schemes are not IND-CPA.

False. Padding in RSA encryption introduces randomness to avoid deterministic output. (It should not be confused with padding in symmetric schemes.)

Right column:

By definition, signatures provide integrity, but encryption doesn't.

(f) (4 points) Which of the following modes provides a guarentee of IND-CPA when properly used? (Mark **all** which apply.)

- ● One-Time Pad
- ● CBC
- ○ ECB
- ● CTR

Solution: ECB is deterministic, so it doesn't provide IND-CPA security.

As shown in lecture, CBC and CTR mode provide IND-CPA security if used correctly (random, unpredictable IVs/nonces). One-time pads provide IND-CPA security if the key (pad) is never reused.

(g) (4 points) Which of the following modes provides an integrity guarente? (Mark **all** which apply.)

- ○ One-Time Pad
- ○ CBC
- ○ ECB
- ○ CTR

Solution: All of these are encryption schemes, which only provide confidentiality. They do not provide any protection against an attacker who tampers with the message.

(h) (4 points) Which of the following make offline dictionary attacks harder? (Mark **all** which apply.)

- ● Slower hash functions
- ● Password Salt
- ○ Faster hash functions
- ● Having users select high entropy passwords

Solution: Slower hashes make dictionary attacks harder because the attacker must spend more time on each guess.

Faster hashes make dictionary attacks easier, not harder, because the attacker can spend less time on each guess.

Salting makes dictionary attacks harder, because it forces the attacker to perform one dictionary attack per user (instead of one dictionary attack for all users).

Users selecting high-entropy passwords makes it harder for the attacker to guess their password. The attacker will have to try a larger dictionary, which makes the dictionary attack harder.

(i) (0 points) I am "Outis"?

- ○ Yes
- ● No

Solution: Some mysteries will always be.

Just-for-fun/attendance question. No relevant course content here.

Problem 2 ***A Random Attempt at a Random Number Generator*** **(16 points)** Consider the following pseudo-code for a pRNG which has Seed, Generate, and Reseed functions. Generate generates 32b values, and the LameRNG uses two cryptographic primitives, a SecureHash (which produces a 256b value), and SecureEncrypt(M,key), a secure block cipher operating on 32b blocks and which uses a 256b key

```
State = {key, ctr}

Seed (entropy) {
   Key = SecureHash(entropy)
   ctr = 0
}

Generate() {
   Return SecureEncrypt(ctr++, key)
}

Reseed(entropy) {
   Key = SecureHash(entropy)
}
```

(a) (4 points) Assume that the attacker doesn't know the key and it is well seeded with entropy. Will generate() produce values that an attacker can't predict (appear random to the attacker) for at least the first 10 outputs?

Solution: Yes. Without knowing the key, the attacker can't predict the output of the secure block cipher.

Note that the block cipher is deterministic, but this doesn't help the attacker, because the counter is being incremented each time the block cipher is called, so the same input is never passed into the block cipher twice.

(b) (4 points) How many times can generate be called before it begins to repeat?

Solution: 2^{32}. The block cipher operates on 32-bit blocks, so there are 2^{32} possible values of the counter. Once all possible counter values have been used, they will start to repeat, and since the block cipher is deterministic, using the same counter twice will result in repeated output.

(c) (4 points) Does this algorithm provide rollback resistance?

Solution: No. If the attacker compromises the internal state (key and counter), they can compute previous pRNG outputs. (Specifically, they calculate `SecureEncrypt(ctr, key)`, decrementing `ctr` to deduce past output.)

(d) (4 points) There is a bug in Reseed(). Fix it:

Solution: `Key = SecureHash(entropy || key)`

Reseed should preserve the existing entropy in the pRNG's current state. The broken implementation resets `key` to only depend on the new entropy supplied in `Reseed`. The fixed implementation makes the new state (`key`) dependent on both the previous state (the old `key`) and the new entropy supplied.

Problem 3 ***Lets Make a Hash of Things*** **(20 points)**

Consider the following small python program designed to select 10 "random" lines from a file and print those out. The `time.time()` function is assumed to be super-precise, measuring current time with nanosecond resolution, so that if you call it multiple times you will get different values each time. As a reminder % is the old-school python string format operation, and Nick is a bit Old Skool when it comes to Python (so `"%s-%i" % ("foo", 32)` will return the string "foo-32"), and `digest()` outputs the sha256 hash of the string as a 32 byte array which, for the comparison operators < and >, is simply a 256b number.

```
1  #!/usr/bin/env python
2
3  import hashlib, sys, time
4
5  hashes = {}
6
7  for line in sys.stdin:
8      for x in range(10):
9          tmp = "%s-%i" % (line, x)
10         h = hashlib.sha256(tmp).digest()
11         if x not in hashes or hashes[x][0] > h:
12             hashes[x] = (h, line)
13
14 for x in range(10):
15     print hashes[x][1]
```

For all the following questions consider it operating on a sample input file consisting of 100 unique and random lines, 99 of which appear only once and one which appears 10,000 times.

(a) (4 points) When this program selects 10 random lines, can it ever select the same line multiple times? Why or why not?

Solution: Yes it can. Each of the 10 runs is independent.

First, let's describe what this program is doing in English.

1. Takes every line of the file and concatenate `0` to the end of the line.
2. Hash every line (with `0` concatenated).
3. Output the line (and its hash) that resulted in the highest hash.
4. Repeat the above steps 9 times, concatenating a different digit (`1` to `9`) each time instead of `0`.

Note that the output of a hash is effectively unpredictable, so for each of the 10 repetitions of steps 1-3, we are effectively choosing one random line from the file. If we change just one digit at the end of the line (the digit we concatenated), the hash output will be totally different, so each of the 10 repetitions is effectively independent.

In other words, it's possible that `<Line 1> || 0` hashes to a higher value than `<Line n> || 0` for all other `n`. At the same time, it's possible that you repeat this process and see that `<Line 1> || 1` hashes to a higher value than `<Line n> || 1` for all other `n`. Thus you would output Line 1 twice.

(b) (4 points) What is the probability that the first output is the line which repeats 10,000 times?

Solution: 1 in 100

As described in the previous part, the first output is the result of performing steps 1-3 with `0` concatenated.

Because hashes are deterministic, hashing the same input 10,000 times results in the same output 10,000 times. This means when you hash every line (with `0` concatenated), you will get 100 distinct hash outputs, because there are 100 distinct inputs. (In other words, the line repeated 10,000 times results in the same hash output 10,000 times.)

From these 100 distinct hash outputs, we select the highest hash. This is effectively a random choice–each hash could be selected with equal probability.

Thus the probability that we choose the hash that was repeated is 1 in 100.

Note that since we are choosing the highest distinct hash, there is no advantage that one of the hashes appeared multiple times. For example, consider a box with slips labeled 1, 2, 2, 2, 2, 3, 4, 5. If you're choosing the slip with the highest hash, it doesn't matter that 2 appears multiple times. You're choosing between H(1), H(2), H(3), H(4), and H(5) (the only distinct values after hashing), each with equal probability to be the highest value.

(c) (4 points) What is the probability that the first output is the line which repeats 10,000 times, if line 9 is changed to `tmp = "%s-%s" % (line, time.time())`?

Solution: 10,000 out of 10,099. Since now it is random but not unique.

Because of the modification, the first output is now generated by hashing all 10,099 lines of the file, *with a different random salt on each line*, and picking the line that resulted in the highest hash value.

Because of the random salt, the hashes are different for every single line, even the repeated lines, so you're effectively picking one line from the file uniformly at random.

There are 10,000+99=10,099 lines, and 10,000 of them are the repeated line, so the probability of choosing the repeated line if you pick any line at random is 10,000 out of 10,099.

(d) (4 points) Consider a version which changes line 11 to `if x not in hashes or hashes[x][0] < h:`. Both the original program and this version are run on an input file containing 100 distinct lines. What is the probability that both versions output the same first line?

Solution: 0. Guaranteed different.

In the original program, to get the first output, you hash all lines of the file (concatenated with `0`) and output the line that resulted in the *highest* hash value.

In the modified program, to get the first output, you again hash all lines of the file (concatenated with `0`) and output the line that resulted in the *lowest* hash value.

Since there are at least 2 (actually 100) distinct lines in the file, you're guaranteed to have at least two different hash values. One program is outputting the maximum, and the other is outputting the minimum, so they are guaranteed to be different.

(e) (4 points) Consider a version which changes line 9 to `tmp = "%i-%s" % (x, line)` Both the original program and this version are run on an input file containing 100 distinct lines. What is the probability that both versions output the same first line?

Solution: 1 in 100.

In the original program, to get the first output, you hash all lines of the file (*concatenated* with `0`) and output the line that resulted in the *highest* hash value.

In the modified program, to get the first output, you hash all lines of the file (*prepended* with `0`) and output the line that resulted in the *highest* hash value.

In both programs, you are choosing one line effectively at random from the set of all 100 distinct lines. The two programs are independent, because the difference in hash input (prepending vs. concatenating) will cause completely different

hash outputs. Thus the probability that you choose the same line twice is 1 in 100.

Problem 4 ***Reasoning About Memory Safety*** **(20 points)**

The following code takes two strings as arguments and returns a pointer to a new string that represents their concatenation:

```
1  char *concat(char s1[], char s2[], int n)
2  {
3    int i, j;
4    int len = strlen(s1) + n;
5    char *s;
6    s = malloc(len);
7    if (!s) return NULL;
8    for (i=0; s1[i] != '\0'; ++i)
9      s[i] = s1[i];
10   for (j=0; s2[j] != '\0' && j < n; ++j)
11     s[i+j] = s2[j];
12   s[i+j] = '\0';
13   return s;
14 }
```

The function is intended to take two strings and return a new string representing their concatenation of the first string with the first n characters of the second string. If a problem occurs, the function's expected behavior is undefined.

(a) For the three statements assigning array elements, write down `Requires` predicates that must hold to make the assignments memory-safe:

1.
```
/* "Requires" for line 9:
 *   s1 != NULL && s != NULL && i >= 0 &&
 *    i < size(s1) && i < size(s)  [same as i < n]
 */
s[i] = s1[i];
```

2.
```
/* "Requires" for line 11:
 *   s2 != NULL and s != NULL && j >= 0 && i+j >= 0 &&
 *    j < size(s2) && i+j < size(s)
 */
s[i+j] = s2[j];
```

3.
```
/* "Requires" for line 12:
 *   s != NULL && i+j >= 0 && i+j < size(s)
 *
 */
s[i+j] = '\0';
```

(b) Here is the same code again, with more space between the lines. Indicate changes (new statements or alterations to the existing code, plus a relevant precondition for calling the function) necessary to ensure memory safety. Do *not* change the types of any of the variables or arguments.

```
/* Precondition:
 * s1 != NULL && s2 != NULL && n >= 0
 */

char *concat(char s1[], char s2[], unsigned int n)

{

  unsigned int i, j;

  unsigned int len = strlen(s1) + n;

  char *s;

  s = malloc(len);

  if (!s) return NULL;

  for (i=0; s1[i] != '\0'; ++i)

    s[i] = s1[i];

  for (j=0; s2[j] != '\0' && j < n; ++j)

    s[i+j] = s2[j];

  s[i+j] = '\0';

  return s;

}
```

Problem 5 ***Alternate Feedback*** **(24 points)**

The following is a diagram of the FFM (F*ed Feedback Mode) block cipher mode of encryption. We assume that the block cipher is a secure block cipher with a 128b block size and key size. Yes, indeed, the initial block encrypts the key with itself...

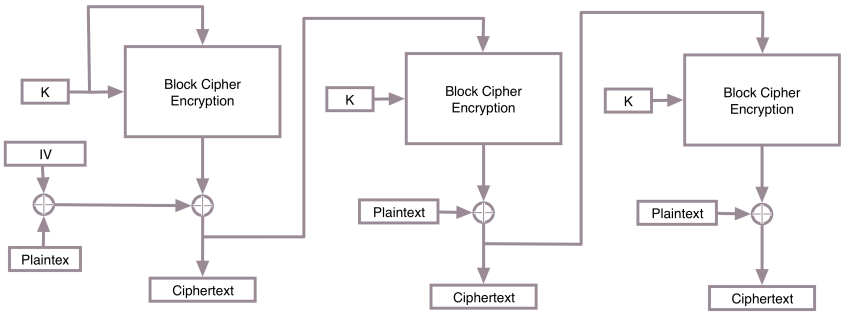
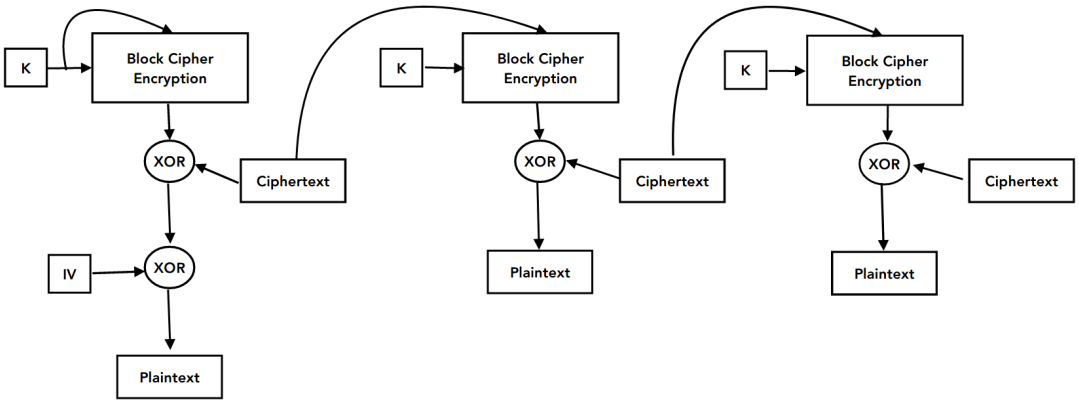


Figure 1: FFM Encryption Mode

(a) (4 points) Draw what the decryption mode will have to look like

Solution:



(b) (4 points) If you reuse the IV for two secret messages, M and M', both using the same key, producing two ciphertexts C and C' seen by the eavesdropper, what can the eavesdroper learn? Assume that the first bit of M and M' are different but the rest of the bits may or may not be the same.

Solution: M xor M' for the first block.

All the indices of matching bits in the first blocks of M and M'. The encryption of K, xor'd with the same IV will not change. If you xor the first blocks of both ciphertexts, the result will be 0 wherever the bits match. This is not true for other blocks, since the first block of ciphertext is fed into AES-encryption to

get the next ciphertext block, which will be different for the two cases.

(c) (4 points) If the first bit of the ciphertext is corrupted in transmission after the encryption is complete and then decrypted, which bits of the decrypted plaintext will be corrupted? (Hint: which decrypted blocks are affected by the first block of ciphertext)

Solution: The first bit in the ciphertext, and the second block.

First bit of decrypted block 1 and all of decrypted block 2.

(d) (4 points) Can this encryption algorithm be parallelized?

○ Yes ● No

(e) (4 points) Can the decryption be parallelized?

● Yes ○ No

(f) (4 points) Is this IND-CPA? Why or why not? (Hint: For IND-CPA, the game can progress multiple times with the same key but a different IV each time and the adversary should still not be able to distinguish which of the two messages is encrypted.)

Solution: It is NOT, since it is E(K,K) xor IV xor M for the first block. Since the IV is public, the eavesdropper can turn this into E(K,K) xor M. Which means the second time through, query with M and M' != M.

No. Suppose you encrypt M twice with the same key but different IV. The first blocks of the the ciphertexts will be:

E(K, K) xor IV xor P1

E(K, K) xor IV2 xor P1

An eavesdropper can xor these together and get IV xor IV2. Since both IV and IV2 are public, this would indicate that the first blocks of plaintext in both messages are the same.

Problem 6 ***The 68x Architecture*** **(26 points)**

Ben Bitdiddle, hack extrodinare, observes that the x86 architecture, where the stack grows down, makes for particularly easy to exploit buffer overflow attacks since a local variable in a buffer grows up to overwrite the saved return address on the stack.

So he proposes the `68x` which effectively flips the logic. Rather than having the stack grow down, the 68x has the stack grow up.

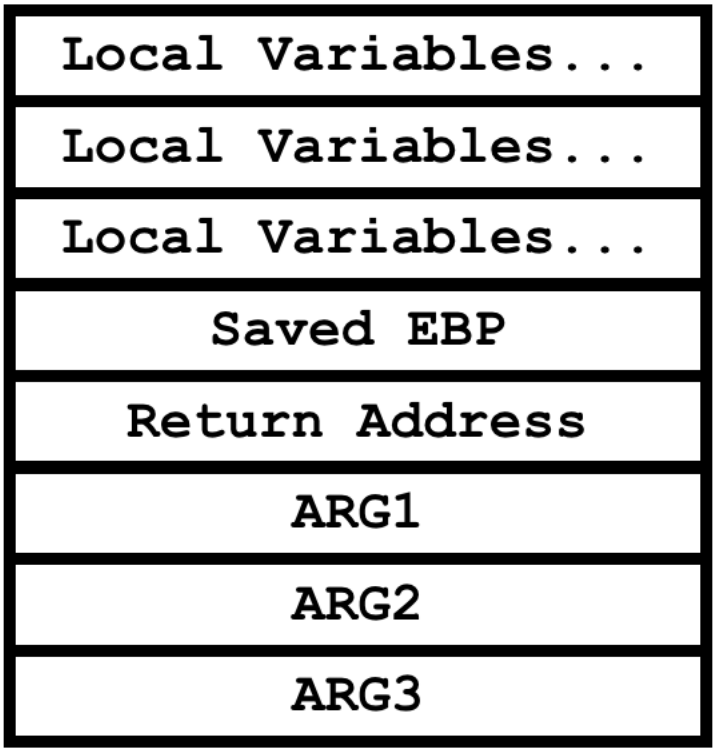


Figure 2: the 68x call frame

The idea is that since buffers write up, by placing the saved return address below a vulnerable buffer the attacker can't overwrite the return address.

Keeping the "upside down x86" theme, if there is a stack canary, it is located between the saved EBP and the local variables on the stack and the stack canary, if it exists, is a random 64b value. 68x is also a 32b architecture and, if ALSR is enabled, it needs to align libraries such that each library starts with the lower 16b of its address as all 0s and all libraries need to be located at an address higher than `0x7FFFFFFF`.

Consider the following simple program.

```
void vuln1(){
   char buffer[16];
   gets(buffer);
}
```

(a) (6 points) Is the simple program exploitable on 68x with a basic stack overflow when the compiler doesn't use stack canaries? Why or why not? (Hint: What does the stack look like when you call gets())

Solution: Yes: You don't overwrite the return address of vulnerable(), but you do end up overwriting the return address of gets()

(b) (4 points) Can the current location of the stack canary prevent an attacker from changing the return address? Why or why not?

Solution: No: You can overwrite the saved return address because its before the canary.

(c) (4 points) Can an attacker ever hope to "brute force" a stack canary on 68x? Why or why not?

Solution: No: 2^{64} is just way too big.

(d) (6 points) The attacker needs to either know or guess the location of a library when attacking ALSR using return oriented programming (ROP). Under what conditions does 68x prevent the attacker from using this technique? Assume that the target program quickly restarts after it crashes.

Solution: It doesn't. This is only 2^{15} entropy, so you can just brute force ALSR

(e) (6 points) Consider this simple program

```
void vuln(){
   char buf[256];
   fgets(stdin,buf,8);
   printf(buf);
}
```

The attacker wishes to determine the state of the stack canary, the function `vuln` has to allocate 256 bytes on the stack for buf and no other space at the point when printf is called. Can the attacker provide an input that will cause the printf to print the value of the stack canary? Why or why not? (Hint: What does the call stack look like when you call printf? What does printf think are the arguments to be printed?)

Solution: The maximum the attacker can put is "%d%d%d", which isn't enough to read out the stack canary.

Does your answer change if we replace `char buf[256];` with `char *buf = malloc(256);`?

Solution: Yeup. Now you can get the canary to print.