Weaver
Fall 2017

CS 161
Computer Security

Midterm 1

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

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

SIGN your name: ____

PRINT your class account login: `cs161-`______ and SID: _________________________

Your TA's name: ____

Your section time: ____

Exam # for person sitting to your left: _______________

Exam # for person sitting to your right: _______________

You may consult one sheet of paper (double-sided) of notes. You may not consult other notes, textbooks, etc. Calculators, computers, and other electronic devices are not permitted.

You have 80 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. Parts of the exam will be graded automatically by scanning the **bubbles you fill in**, so please do your best to fill them in somewhat completely. Don't worry—if something goes wrong with the scanning, you'll have a chance to correct it during the regrade period.

If you have a question, raise your hand, and when an instructor motions to you, come to them to ask the question.

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

Question:	1	2	3	4	5	6	Total
Points:	36	16	20	20	24	26	142
Score:							

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: CTR you lose against known plaintext, but chosen ciphertext doesn't help here. CFB doesn't lose completely, you do lose IND-CPA tho. And nick's spirit is well known on this issue.

(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

(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 positive rate will increase.
- ● Your false negative rate will decrease.
- ○ Your false positive rate will decrease.

Solution: This is a defense in depth approach, as an attacker needs to get through both. Which reduces your false negative rate. But it increases your false positive rate.

(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.
- ● An attacker can use Return Oriented Programming
- ○ Buffer overflows are no longer exploitable
- ● Format-String vulnerabilities may still be exploitable

Solution: You can still use ROP, but can't write code into memory to execute.

(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: Signatures provide integrity, encryption doesn't. You need to pad for IND-CPA but padding isn't just involivng 0s.

(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 doesn't all others good

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

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

Solution: None do, sorry...

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

● Slower hash functions

○ Faster hash functions

● Password Salt

● Having users select high entropy passwords

Solution: Faster hashes kindsa makes it easier...

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

○ Yes

● No

Solution: Some mysteries will always be.

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

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

Solution: 2^{32}

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

Solution: No.

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

Solution: Key = SecureHash(entropy —— key)

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.

```
#!/usr/bin/env python

import hashlib, sys, time

hashes = {}

for line in sys.stdin:
    for x in range(10):
        tmp = "%s-%i" % (line, x)
        h = hashlib.sha256(tmp).digest()
        if x not in hashes or hashes[x][0] > h:
            hashes[x] = (h, line)

for x in range(10):
    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 is independent.

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

> **Solution:** 1 in 100

(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.

(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.

(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.

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.

```c
/* 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 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.

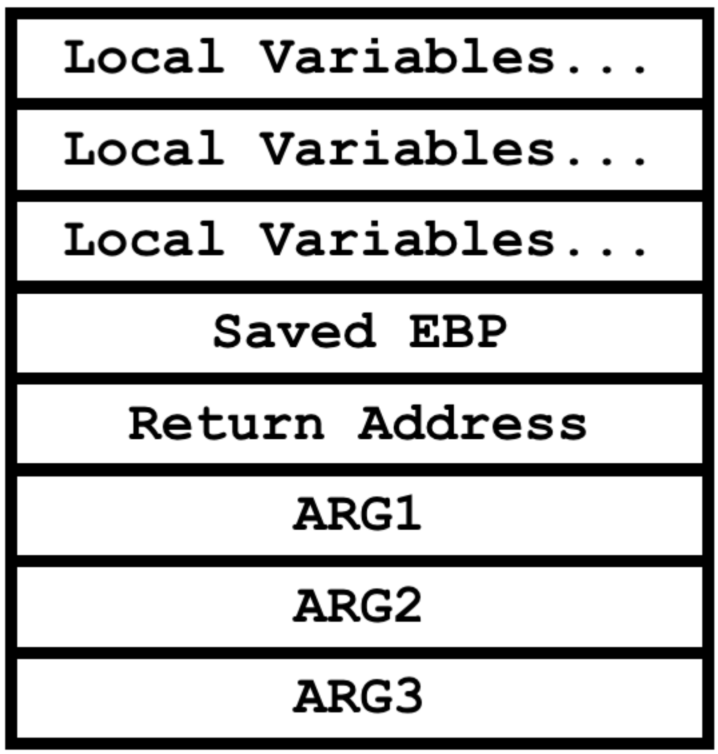

Figure 2: the 68x call frame

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.