Popa & Weaver Spring 2019	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 academic misconduct will be reported to the Center for Student Conduct and may further result in partial or complete loss of credit. I am also aware that Nick Weaver in particular takes cheating personally and, like the Hulk®, you don't want to see him angry.

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, textbooks, etc. Calculators, computers, and other electronic devices are not permitted.

Bubble every item completely! Avoid using checkmarks, Xs, writing answers on the side, etc.. 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 any number of choices (including none or all).

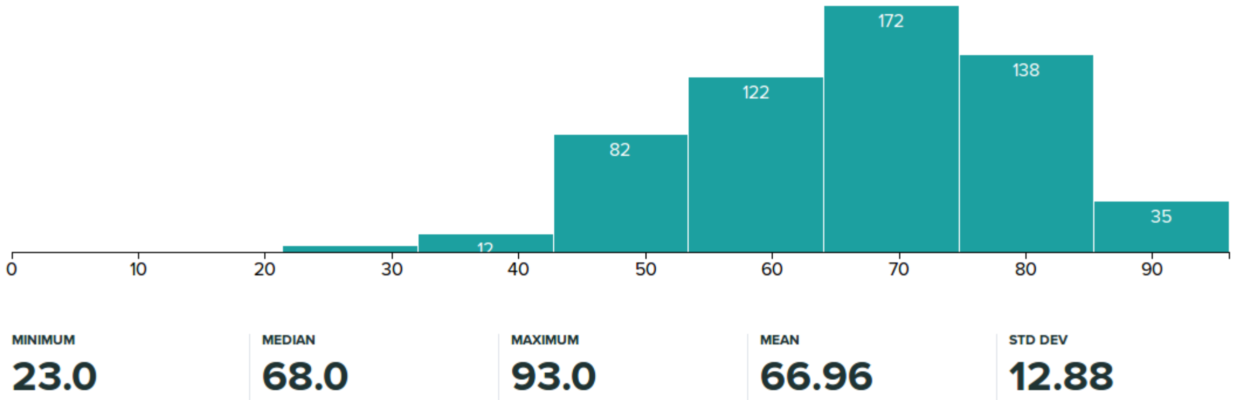
- ■ You can select
- ■ multiple squares (completely filled).

If you think a question is ambiguous, please come up to the front of the exam room to the 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 8 questions, of varying credit (96 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

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

Grade distribution (out of 96 points):



Problem 1 *Potpourri Question* (16 points)

(a) True or False: Unlike CTR mode, CBC offers integrity against flipping bits of the ciphertext.

○ True ● False

Solution: Bit flipping in CBC can cause different decryption result.

(b) True or False: ASLR helps prevent buffer overflow attacks by randomizing the relative position of a buffer with respect to the overwritable return instruction pointer on the stack.

○ True ● False

(c) True or False: ASLR helps prevent buffer overflow attacks by randomizing the relative order in which function stack frames are placed on the stack.

○ True ● False

Solution: ASLR randomizes the start of different memory sections, but does not affect the order of each function's stack frame.

(d) True or False: It is possible to use the `ret2esp` attack from Project 1 when W^X is enabled.

○ True ● False

Solution: As described, the ret2esp attack is not possible because the shellcode can only be placed on the stack. W^X/NX/DEP will prevent the shellcode from executing, so the attack will fail.

(e) True or False: Sandboxing different parts of an application can help reduce the size of the TCB.

● True ○ False

(f) True or False: Symmetric key encryption is faster than asymmetric key encryption.

● True ○ False

(g) Let m be a message, let E_k be any IND-CPA encryption scheme and MAC_k be any secure MAC function. Let k be a randomly generated key. Write $C = E_k(m)$.

True or False: If an eavesdropper sees $C \| \text{MAC}_k(C)$, the message m is still confidential.

○ True ● False

Solution: Nope, reuses key k! It is **incorrect** in general to reuse a key for two different purposes, as discussed in the 2/19 and 2/21 lectures.

(h) Mallory is a man-in-the-middle attacker, but Alice and Bob want to send messages to each other without her interference. Which of the following properties **alone** is enough to ensure that Mallory can **neither read nor tamper** with any of their messages?

- [ ] Confidentiality
- [ ] Authenticity
- [ ] Polytime Hardness
- [ ] Integrity
- [ ] Availability
- [x] None of the above

Solution: None of the properties alone are enough.

Problem 2 *Greetings from Mallory* **(9 points)**

The following program has two security-critical vulnerabilities. APPENDIX: See the Appendix for a list of C functions.

```
1  void get_name(char *prompt, char *greeting) {
2    printf(prompt);
3    int fd = 0;  // stdin
4    char *buf = greeting + strlen(greeting);  // remaining buffer
5    size_t count = sizeof(greeting) - strlen(greeting);  // size left
6    read(fd, buf, count);
7  }
8
9  int main() {
10   char prompt[] = "Please enter your name:\n";
11   char greeting[64] = "Welcome back, ";
12   get_name(prompt, greeting);
13   printf(greeting);
14 }
```

Identify the two security-critical vulnerabilities in the code. For each vulnerability, provide the line number and a short explanation. (GRADING NOTE: You will receive six points if you find one vulnerability, and nine points if you find both vulnerabilities.)

(a) **Vulnerability 1:**

◇ Line number: _______

◇ Explanation: (20 words max)

(b) **Vulnerability 2:**

◇ Line number: _______

◇ Explanation: (20 words max)

Solution:

- Vulnerability 1:

  Line 5

  `sizeof(greeting)` evaluates to `sizeof(char *)`, which is 4 and not 64, so `count` underflows and becomes a large unsigned number. The read operation can then overflow past the end of the `greeting` buffer.

  This can be exploited by overwriting the saved return address of function `main` and hijacking the control-flow of the program.

- Vulnerability 2:

  Line 13

  String format vulnerability, since contents of the `greeting` argument are controlled by the attacker, who can insert format modifiers `%s`, `%n`, etc.

  A string format vulnerability is very dangerous and can actually allow arbitrary code execution through special use of `%n` and other formats.

- Another issue (partial credit):

  Another issue with this program is that the `greeting` string is not guaranteed to be null terminated after the read operation. This allows the `printf` function on line 13 to read past the end of the `greeting` buffer. On this particular program, this is not security-critical, because after the `greeting` buffer it will find the `prompt` buffer, which is null terminated and doesn't give any new information to the attacker.

Problem 3 ***Prince of Security*** **(8 points)**

(a) Rather than using a password manager, you decide to hide your passwords under the directory `old-tax-returns/old-things/not-secret/passwords`, reasoning that it is secure because hackers won't be able to find them. Which security principle does this violate?

○ Least privilege ○ Consider human factors

● Shannon's maxim ○ Know your threat model

(b) At night, you cannot enter Etcheverry without special cardkey access. However you can get around this by going to the second floor of Soda, and then using your cardkey to open the Etcheverry-Soda door on the second floor. Which security principle does this violate?

● Ensure complete mediation ○ Consider human factors

○ Shannon's maxim ○ Least privilege

(c) You enjoy CS 161 and decide to become the head TA! The front desk hands you physical keys: some access the printing room and some access a closet full of exam questions. You give away only the keys which access the printing room to the TAs in charge of printing discussion worksheets. Which security principle did you consider?

○ Ensure complete mediation ○ Division of trust

○ Design in security from the start ● Least privilege

(d) In certain government agencies, employees are required to use government-approved phones for work. Some employees find these phones too difficult to use, so they do work on their personal phones instead. Which security principles does this violate?

○ Ensure complete mediation ○ Division of trust

○ Least privilege ● Consider human factors

Problem 4 ***AES-CBC-STAR*** **(13 points)**

Let E_k and D_k be the AES block cipher in encryption and decryption mode, respectively.

(a) We invent a new encryption scheme called AES-CBC-STAR. A message M is broken up into plaintext blocks $M_1, \ldots, M_n$ each of which is 128 bits. Our encryption procedure is:

$$C_0 = \texttt{IV} \text{ (generated randomly)},$$
$$C_i = E_k(C_{i-1} \oplus M_i) \oplus C_{i-1}.$$

where $\oplus$ is bit-wise XOR.

◇ Write the equation to decrypt M_i in terms of the ciphertext blocks and the key k.

> **Solution:** $M_i = D_k(C_i \oplus C_{i-1}) \oplus C_{i-1}$.

(b) Mark each of the properties below that AES-CBC-STAR satisfies. Assume that the plaintexts are 100 blocks long, and that $10 \le i \le 20$.

- ☐ Encryption is parallelizable.
- ■ Decryption is parallelizable.
- ☐ If C_i is lost, then C_{i+1} can still be decrypted.
- ☐ If we flip the least significant bit of C_i, this always flips the least significant bit in P_i of the decrypted plaintext.
- ☐ If we flip a bit of M_i and re-encrypt using the same IV, the encryption is the same except the corresponding bit of C_i is flipped.
- ■ If C_i is lost, then C_{i-1} can still be decrypted.
- ■ If C_i is lost, then C_{i+2} can still be decrypted.
- ■ If C_i is lost, then C_{i-2} can still be decrypted.
- ☐ If we flip the least significant bit of C_i, this always flips the least significant bit in P_{i+1} of the decrypted plaintext.
- ☐ It is not necessary to pad plaintext to the blocksize of AES when encrypting with AES-CBC-STAR.

(c) Now we consider a modified version of AES-CBC-STAR, which we will call AES-CBC-STAR-STAR. Instead of generating the IV randomly, the challenger uses a list of random numbers which are public and known to the adversary. Let $\texttt{IV}_i$ be the IV which will be used to encrypt the ith message from the adversary.

◇ Argue that the adversary can win the IND-CPA game.

> **Solution:** Adversary sends two arbitrary (unequal but equal length), one-block messages (M, M') as the challenge. The resulting ciphertext is either $C_0 = \texttt{IV}_0 || E_k(\texttt{IV}_0 \oplus M) \oplus \texttt{IV}_0$ or $C_0 = \texttt{IV}_0 || E_k(\texttt{IV}_0 \oplus M') \oplus IV_0$.
>
> Next the adversary sends $\texttt{IV}_1 \oplus \texttt{IV}_0 \oplus M$. The resulting ciphertext is $C_1 = \texttt{IV}_1 || E_k(\texttt{IV}_1 \oplus (\texttt{IV}_0 \oplus \texttt{IV}_1 \oplus M)) \oplus \texttt{IV}_1$, which simplifies to $\texttt{IV}_1 || E_k(\texttt{IV}_0 \oplus M) \oplus \texttt{IV}_1$. If the second block of $C_1 \oplus \texttt{IV}_1$ equals the second block of $C_0 \oplus \texttt{IV}_0$, then the challenger encrypted M. Otherwise the challenger encrypted M'. Hence we break IND-CPA with advantage significantly above $\frac{1}{2}$ (in fact such an adversary wins all the time).

An alternative solution is to send the challenger ciphertexts $M = \texttt{IV}_1$ and $M' =$ anything else. If the challenger encrypts M, the message received is $E_k(0) \oplus \texttt{IV}_1$. Then for the second message, send $\texttt{IV}_2$. If the output ciphertext $\oplus \texttt{IV}_1 \oplus \texttt{IV}_2$ equals the challenge ciphertext, then the challenger encrypted M. Otherwise they encrypted M'.

Problem 5 ***Extreme conditioning*** **(9 points)**

Consider the following code:

```
1  int my_strcmp(char *s1, char *s2) {
2      size_t i = 0;
3      while (s1[i]) {
4          /** part b **/
5          if (s1[i] != s2[i]) {
6              break;
7          }
8          i++;
9      }
10     char uc1 = *s1, uc2 = *s2;
11     if (uc1 < uc2) return -1;
12     return uc1 > uc2;
13 }
```

(a) Consider the preconditions necessary to ensure memory safety. What is required about null termination and length of the strings?

⋄ Write **at most two** preconditions, of **at most ten words each**.

> **Solution:** (1) s1 must be null terminated
>
> (2) The length of s1 also cannot be greater than the length of s2 or s2 must be null terminated, otherwise we would read past the end of s2.

(b) State one invariant at **line 4** about `s1` that is about memory safety. Do not include an invariant which is already a precondition.

⋄ Write this invariant.

> **Solution:** for all x in [0, i], s1[x] ! = '\0' OR 0 <= i < strlen(s1) [these two are equivalent]

Problem 6 *Please, Just Use HMAC* (8 points)

Alice and Bob are partners struggling through their CS 161 project, and need to share code with one another, but their only option is to pass messages through an insecure server in Soda. They are afraid another student, Mallory, might read or tamper with the messages.

They have already established public-keys (P_A and P_B), secret keys (S_A and S_B) and two shared symmetric keys (k and k'). Using these, the SHA3 cryptographic hash function (SHA3), and an IND-CPA secure symmetric-key encryption (Enc_k), Alice proposes a set of ways to send her messages (M) to Bob. Note that || denotes the concatenation operation.

(a) Mark which of her following proposals provide confidentiality and allow Bob to retrieve the message M in the presence of only passive adversaries. (Select all that apply.)

- ☐ M || SHA3(M)
- ■ $\text{Enc}_k(M)$
- ☐ SHA3(M || k')
- ☐ M || SHA3(M || S_A)
- ☐ M || SHA3(M || P_B)
- ☐ $\text{Enc}_k(M)$ || SHA3(M || k')

Solution:

Any protocol that provides the plaintext (M) in the clear does not provide confidentiality. Enc is an IND-CPA encryption function, as is by definition confidential.

Since cryptographic hash functions are deterministic, they do not provide confidentiality. In particular, an attacker can tell if the same message is sent twice.

(b) Mark which of her following proposals provide integrity. (Select all that apply.)

- ☐ M || SHA3(M)
- ☐ $\text{Enc}_k(M)$
- ☐ SHA3(M || k')
- ☐ M || SHA3(M || S_A)
- ☐ M || SHA3(M || P_B)
- ■ $\text{Enc}_k(M)$ || SHA3(M || k')

Solution: Any proposal that does not include any secret information cannot provide integrity, since the entire ciphertext can be recomputed by the adversary Mallory. This includes proposals that use P_B, since this is a *publicly*-known key.

H(M || K) fails to provide integrity, since the original message is not recoverable (Bob cannot invert the cryptographic hash function). Bob does not have access to Alice's secret key, and can never compute H(M || S_A) to verify that M was not tampered with.

Lastly, Enc_K(M) fails to provide integrity even though Mallory doesn't know K: she doesn't have to create a valid encryption to tamper with the original. While tampered messages will likely decrypt to random bits, this is still often useful (i.e., Alice is sending a new random key).

Problem 7 ***ElGamal and friends*** **(15 points)**

Bob wants his pipes fixed and invites independent plumbers to send him bids for their services (*i.e.*, the fees they charge). Alice is a plumber and wants to submit a bid to Bob. Alice and Bob want to preserve the confidentiality of Alice's bid, but the communication channel between them is insecure. Therefore, they decide to use the ElGamal public key encryption scheme in order to communicate privately.

Instead of using the traditional version of the ElGamal scheme, Alice and Bob use the following variant. As usual, Bob's private key is x and his public key is $\mathsf{PK} = (p, g, h)$, where $h = g^x \bmod p$. However, to send a message M to Bob, Alice encrypts M as $\mathsf{Enc}_{\mathsf{PK}}(M) = (s, t)$, where $s = g^r \bmod p$ and $t = g^M \times h^r \bmod p$, for a randomly chosen r.

(a) Consider two distinct messages m_1 and m_2. Let $\mathsf{Enc}_{\mathsf{PK}}(m_1) = (s_1, t_1)$ and $\mathsf{Enc}_{\mathsf{PK}}(m_2) = (s_2, t_2)$. For the given variant of the ElGamal scheme, which of the following is true?

- ○ $(s_1 + s_2 \bmod p, \quad t_1 + t_2 \bmod p)$ is a possible value for $\mathsf{Enc}_{\mathsf{PK}}(m_1 + m_2)$.
- ● $(s_1 \times s_2 \bmod p, \quad t_1 \times t_2 \bmod p)$ is a possible value for $\mathsf{Enc}_{\mathsf{PK}}(m_1 + m_2)$.
- ○ $(s_1 \times s_2 \bmod p, \quad t_1 \times t_2 \bmod p)$ is a possible value for $\mathsf{Enc}_{\mathsf{PK}}(m_1 \times m_2)$.
- ○ $(s_1 + s_2 \bmod p, \quad t_1 + t_2 \bmod p)$ is a possible value for $\mathsf{Enc}_{\mathsf{PK}}(m_1 \times m_2)$.
- ○ None of these

(b) In order to decrypt a ciphertext (s, t), Bob starts by calculating $q = ts^{-x} \bmod p$. Assume that the message M is between 0 and 1000. How can Bob recover M from q?

> **Solution:** If Bob knows the possible set of messages, then he can pre-compute a lookup table for values of $q = g^M \bmod p$.

(c) Explain why Bob cannot efficiently recover M from q if M is randomly chosen such that $0 \le M < p$.

> **Solution:** Requires solving the discrete log mod$p$, which is thought to be computationally hard.

(d) Suppose Alice sends Bob a bid $M_0 = 500$, encrypted under Bob's public key. We let $C_0 = (s, t)$ be the ciphertext here.

Mallory is an active man-in-the-middle attacker who knows Alice's bid is $M_0 = 500$. Mallory wants to replace Alice's bid with $M_1 = 999$. To do that, Mallory intercepts C_0 and replaces it with another ciphertext C_1. Mallory wishes that when Bob decrypts C_1, Bob sees $M_1 = 999$.

Describe how Mallory creates C_1 in each of the following situations:

1. Mallory didn't obtain C_0, but knows Bob's public key $\mathsf{PK} = (p, g, h)$.

   ⋄ Question: How should Mallory create C_1?

   Solution: Mallory can simply encrypt M of her choice using Bob's public key and replace the ciphertext.

2. Mallory knows Alice's ciphertext C_0, but only knows p and g in Bob's public key $\mathsf{PK} = (p, g, h)$. (That is to say, Mallory does not know h.)

   ⋄ Question: How should Mallory create C_1?

   Solution: Mallory can create $(s', t') = (s, tg^{499}) \pmod{p}$.

Problem 8 ***Canaries Schmanaries*** **(18 points)**

The following code runs on a 32-bit x86 system. **Stack canaries are enabled**, but other memory safety defenses are disabled. As in Project 1, all four bytes of the canary are completely random.

The compiler does not rearrange stack variables. Note the `volatile` keyword on line 1: this means the arguments `s1` and `s2` are loaded from memory whenever referenced by `doit`, instead of being stored in registers. APPENDIX: See the Appendix for a list of C functions.

```
1  void doit(char* volatile s1, char* volatile s2) {
2      char buffer[16];
3      strcpy(buffer, s1);
4      strcpy(s1, s2);
5      printf("%s\n%s\n%s\n", buffer, s1, s2);
6  }
7
8  int main() {
9      char s1[64]; char s2[64];
10     fgets(s1, sizeof s1, stdin);
11     fgets(s2, sizeof s2, stdin);
12     doit(s1, s2);
13 }
```

(a) Which line contains a memory safety vulnerability? What is the name of the vulnerability present on that line?

> **Solution:** Line 3: buffer overflow.

(b) Complete the diagram of the stack, right before line 3. Assume normal (non-malicious) program execution. You do not need to write the values on the stack, only the names. There are no extraneous boxes. As in discussion, the bottom of the page represents the lower addresses.

compiler padding = 0x00000000
main's canary
char s1 [64]
char s2 [64]
s2
s1
saved eip / rip
saved ebp / sfp
doit's canary
char buffer[16]

(c) Now we will exploit the program. There is already shellcode at the address `0xbfffdead`. Using `gdb`, you discovered that the address of `main's canary` is `0xbfffdab4`. Due to a bug in the compiler, you discover that although stack canaries are present, **they are not checked!** Complete the Python script below in order to successfully exploit the program.

NOTE: The Python syntax `'A' * n` indicates that the character `'A'` will be repeated `n` times. The syntax `\xRS` indicates a byte with hex value `0xRS`.

```
s1 = 'A' * ____ + '____________________________________________________' + \
                  '____________________________________________________'

s2 = 'B' * ____ + '____________________________________________________' + \
                  '____________________________________________________'

print s1
print s2
```

Solution:

```
s1 = 'A' * 24 + '\xad\xde\xff\xbf'
s2 = 'anything'
```

Note that there is a slight technical nit, since `fgets` adds a newline and a terminating NUL character. This means that such a solution clobbers the address of `s1`. In practice this is unlikely to be an issue, although one can get around it by writing the original values into `s1` and `s2`. We didn't deduct points from solutions which failed to notice this issue.

(d) Unfortunately, the bug in the previous part was fixed, and now your exploit must successfully bypass the stack canary. As in part (c), there is already shellcode at the address `0xbfffdead` and the address of `main's canary` is `0xbfffdab4`. Complete the Python script below in order to successfully exploit the program.

HINT: You should do the following. Start by using your exploit from the part above. Overwrite the arguments `s1` and `s2` of `doit` to ensure that the second `strcpy` will "fix" the canary. Note that the `main`'s function frame has the same canary as the canary that should appear in `doit`'s function frame. The use of the `volatile` keyword ensures that `s1` and `s2` are passed using their values from the stack. Since your solution should overwrite the pointer `s2`, it does not matter what it originally points to.

```
s1 = 'A' * ____ + '____________________________________________________' + \
                  '____________________________________________________' + \
                  '____________________________________________________'

s2 = 'not needed, see the hint'
```

```
print s1
print s2
```

Solution:

```
s1 = 'A' * 24 + '\xad\xde\xff\xbf'+'\x20\xda\xff\xbf'+'\xb4\xda\xff\xbf'
s2 = 'not needed, see the hint'
print s1
print s2
```

Explanation: after the execution of the first `strcpy` on line 3, our stack looks as follows:

`main's canary`
`char s1[64]`
`char s2[64]`
`s2 = 0xbfffdab4`
`s1 = 0xbfffda20`
`saved eip = 0xbfffdead`
`saved ebp = AAAA`
`doit's canary = AAAA`
`char buffer[16] = A...A`

Even though the canary is messed up after the first `strcpy`, this does not cause the program to exit. Stack canaries are only checked before we exit the function.

Note that we have now overwritten the arguments to `doit`. We have engineered the stack such that `s2 = &main's canary` and `s1 = &doit's canary`. The next `strcpy` on line 4 therefore fixes `doit's` canary. The compiler padding of all `0`s is useful, because it acts as a NUL-terminator for the `strcpy`. It does clobber the last byte of the saved ebp, but that doesn't matter since the shellcode will execute before this becomes a problem.

The exploit works with probability ≈ 63/64, since the canary might have a NUL byte, making it impossible to copy via `strcpy`.

Note that an optimizing compiler might save the value of `s1` in a register in between the two `strcpy` calls, which would prevent this exploit, but the use of the `volatile` keyword prevents this.

Selected C Manual Pages

```
char *fgets(char *s, int size, FILE *stream);
fgets() reads in at most one less than _size_ characters from
_stream_ and stores them into the buffer pointed to by _s_. Reading
stops after an EOF or a newline.  If a newline is read, it is stored
into the buffer.  A terminating null byte ('\0') is stored after the
last character in the buffer.
```

```
int printf(const char *format, ...);
printf() produces output according to the format string _format_.
```

```
ssize_t read(int fd, void *buf, size_t count);
read() attempts to read up to _count_ bytes from file descriptor _fd_
into the buffer starting at _buf_.
```

```
char *strcpy(char *dest, const char *src);
The strcpy() function copies the string pointed to by _src_,
including the terminating null byte ('\0'), to the buffer pointed to
by _dest_.
```

```
size_t strlen(const char *s);
The strlen() function calculates the length of the string _s_,
excluding the terminating null byte ('\0').
```

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