Memory Safety

Question 1 ***Software Vulnerabilities*** ()

For the following code, assume an attacker can control the value of `basket`, `n`, and `owner_name` passed into `search_basket`.

The code includes several security vulnerabilities. **Circle *three* such vulnerabilities** in the code and **briefly explain** each of the three on the next page.

```
1  struct cat {
2      char name[64];
3      char owner[64];
4      int age;
5  };
6  /* Searches through a BASKET of cats of length N (N should be less than 32) and
7   * adopts all cats with age less than 12 (kittens). Adopted kittens have their
8   * owner name overwritten with OWNER_NAME. Returns the number of kittens
9   * adopted. */
10 size_t search_basket(struct cat *basket, int n, char *owner_name) {
11     struct cat kittens[32];
12     size_t num_kittens = 0;
13     if (n > 32) return -1;
14     for (size_t i = 0; i <= n; i++) {
15         if (basket[i].age < 12) {
16             /* Reassign the owner name. */
17             strcpy(basket[i].owner, owner_name);
18             /* Copy the kitten from the basket. */
19             kittens[num_kittens] = basket[i];
20             num_kittens++;
21             /* Print helpful message. */
22             printf("Adopting kitten: ");
23             printf(basket[i].name);
24             printf("\n");
25         }
26     }
27     /* Adopt kittens. */
28     adopt_kittens(kittens, num_kittens); // Implementation not shown.
29     return num_kittens;
30 }
```

1. Explanation:

2. Explanation:

3. Explanation:

4. Explanation:

Describe how an attacker could exploit these vulnerabilities to obtain a shell:

Question 2 ***C Memory Defenses*** ()

Mark the following statements as True or False and justify your solution. Please feel free to discuss with students around you.

1. Stack canaries completely prevent a buffer overflow from overwriting the return instruction pointer.

2. A format-string vulnerability can allow an attacker to overwrite values below the stack pointer

3. An attacker exploits a buffer overflow to redirect program execution to their input. This attack no longer works if the data execution prevention/executable space protection/NX bit is set.

4. If you have a non-executable stack and heap, buffer overflows are no longer exploitable.

5. If you use a memory-safe language, some buffer overflow attacks are still possible.

6. ASLR, stack canaries, and NX bits all combined are insufficient to prevent exploitation of all buffer overflow attacks.

Short answer!

1. What vulnerability would arise if the canary was above the return address?

2. What vulnerability would arise if the stack canary was between the return address and the saved frame pointer?

3. Assume ASLR is enabled. What vulnerability would arise if the instruction **jmp *esp** exists in memory?

Question 3 ***TCB (Trusted Computing Base)*** ()

In lecture, we discussed the importance of a TCB and the thought that goes into designing it. Answer these following questions about the TCB:

1. What is a TCB?

   _____

2. What can we do to reduce the size of the TCB?

   _____

3. What components are included in the (physical analog of) TCB for the following security goals:

   (a) Preventing break-ins to your apartment

   _____

   (b) Locking up your bike

   _____

   (c) Preventing people from riding BART for free

   _____

   (d) Making sure no explosives are present on an airplane

   _____

Question 4 *Canaries Schmanaries* ()

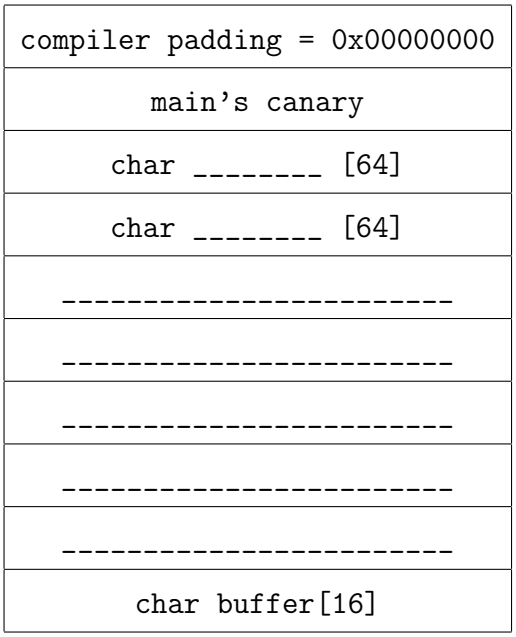
The following code runs on a 32-bit x86 system. **Stack canaries are enabled**, but other memory safety defenses are disabled. As in Project 1, all four bytes of the canary are completely random.

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?

(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.



(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
```