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:

> **Solution:** Line **15** has a fencepost error: the conditional test should be $i < n$ rather than $i <= n$. The test at line **13** assures that **n** doesn't exceed 32, but if it's equal to 32, and if all of the cats in **basket** are kittens, then the assignment at line **19** will write past the end of **kittens**, representing a buffer overflow vulnerability.

2. Explanation:

> **Solution:** At line **14** we are checking if i <= n. i is an unsigned int and n is a signed int, so during the comparison n is cast to an unsigned int. We can pass in a value such as n = -1 and this would be cast to 0xffffffff which allows the for loop to keep going and write past the buffer.

3. Explanation:

> **Solution:** On line **17** there is a call to strcpy which writes the contents of `owner_name`, which is controlled by the attacker, into the owner instance variable of the cat struct. There are no checks that the length of the destination buffer is greater than or equal to the source buffer `owner_name` and therefore the buffer can be overflown.

4. Explanation:

> **Solution:** On line **23** there is a printf call which prints the value stored in the name instance variable of the cat struct. This input is controlled by the attacker and is therefore subject to format string vulnerabilities since the attacker could assign the cats names with string formats in them.

> **Solution:** Some more minor issues concern the **name** strings in **basket** possibly not being correctly terminated with '\0' characters, which could lead to reading of memory outside of **basket** at line **23**.

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

> **Solution:** Each vulnerability could lead to code execution. An attacker could also use the fencepost or the bound-checking error to overwrite the rip and execute arbi-

trary code.

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.

   Solution:

   False, stack canaries can be defeated if they are revealed by information leakage, or if there is not sufficient entropy, in which case an attacker can guess the value. Also, format string vulnerabilities can simply skip past the canary.

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

   Solution:

   True, format string vulnerabilities can write to arbitrary addresses by using a '%n' in junction with a 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.

   Solution:

   True, the definition of the NX bit is that it prevents code from being writable and executable at the same time. An attacker who can write code into memory cannot execute it.

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

   Solution:

   False. Many attacks rely on writing malicious code to memory and then executing them. If we make writable parts of memory non-executable, these attacks cannot succeed. However there are other types of attacks which still work in these cases, such as Return Oriented Programming.

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

Solution:

False, buffer overflow attacks do not work with memory safe languages.

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

Solution:

True, all of these protections can be overcome.

Short answer!

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

Solution:

It doesn't stop an attacker from overwriting the return address. Although if an attacker had absolutely no idea where the return address was, it could potentially detect stack smashing.

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

Solution:

An attacker can overwrite the saved frame pointer so that the program uses the wrong address as the base pointer after it returns. This can be turned into an exploit.

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

Solution:

An attacker can overwrite the return instruction pointer with the address of this command. This will cause the function to execute the instruction one word before the rip. An attacker could place the shellcode after the rip, and have the word before the rip contain a JMP command two words forward.

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

Solution:

1. It is the set of hardware and software on which we depend for correct enforcement of policy. If part of the TCB is incorrect, the system's security properties can no longer be guaranteed to be true. Anything outside the TCB isn't relied upon in any way.
2. Privilege separation and separation of responsibility can help reduce the size of the TCB. You will end up with more components, but not all of them can violate your security goals if they break. The size of the TCB can also be reduced by reducing the application's dependency on third-party components and software.
3. (This list is not necessarily complete)
   (a) the lock, the door, the walls, the windows, the roof, the floor, you, anyone who has a key
   (b) the bike frame, the bike lock, the post you lock it to, the ground
   (c) the ticket machines, the tickets, the turnstiles, the entrances, the employees
   (d) the TSA employees, the security gates, the "one-way" exit gates, the fences surrounding the runway area

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?

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.