Popa & Kao CS 161
Spring 2023 Computer Security Discussion 2

Question 1 ***Softwar*** [illegible] ()

For the following c[illegible] can control the value of basket, n, and owner_name passed into search_bask[illegible]

This code contains [illegible] abilities. **Circle *three* such vulnerabilities** in the code and briefly explain each [illegible] xt page.

```
struct cat {
    char name[64];
    char owner[64];
    int age;
};

/* Searches through a BASKET of cats of length N (N should be less than 32). Adopts all cats with age less than 12 (kittens). Adopted kittens have their owner name overwritten with OWNER_NAME. Returns the number of kittens adopted. */
size_t search_basket(struct cat *basket, int n, char *owner_name) {
    struct cat kittens[32];
    size_t num_kittens = 0;
    if (n > 32) return -1;
    for (size_t i = 0; i <= n; i++) {
        if (basket[i].age < 12) {
            /* Reassign the owner name. */
            strcpy(basket[i].owner, owner_name);
            /* Copy the kitten from the basket. */
            kittens[num_kittens] = basket[i];
            num_kittens++;
            /* Print helpful message. */
            printf("Adopting kitten: ");
            printf(basket[i].name);
            printf("\n");
        }
    }
    /* Adopt kittens. */
    adopt_kittens(kittens, num_kittens); // Implementation not shown.
    return num_kittens;
}
```

1. Explanation:
2. Explanation:
3. Explanation:

Describe how an at ese vulnerabilities to obtain a shell:

Question 2 *Echo, Echo, Echo* ()

Consider the following vulnerable C code:

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

char name[32][illegible]

void echo(voi[illegible]
    char echo[illegible]
    printf("V[illegible] me to echo back?\n");
    gets(echo[illegible]
    printf("%s\n", echo_str);
}

int main(void) {
    printf("What's your name?\n");
    fread(name, 1, 32, stdin);
    printf("Hi, %s\n", name);

    while (1) {
        echo();
    }

    return 0;
}
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all questions.

Q2.1 (5 min) Assume that non-executable pages are enabled so we cannot execute SHELLCODE on stack. We would like to exploit the `system(char *command)` function to start a shell. This function executes the string pointed to by `command` as a shell command. For example, `system("ls")` will list files in the current directory.

Construct an would cause the program to execute the function call `system("sh"` ress of `system` is `0xdeadbeef` and that the address of the RIP of `echo` is our answer in Python syntax (like in Project 1).

Hint: Recall th relies on setting up the stack so that, when the program pops off and jumps et up in a way that looks like the function was called with a particular argu

Q2.2 (6 min) Assume that, in addition to non-executable pages, ASLR is also enabled. However, addresses of global variables are not randomized.

Is it still possible to exploit this program and execute malicious shellcode?

○ (G) Yes, because you can find the address of both `name` and `system`

○ (H) Yes, because ASLR preserves the relative ordering of items on the stack

○ (I) No, because non-executable pages means that you can't start a shell

○ (J) No, because ASLR will randomize the code section of memory

○ (K) ——

○ (L) ——

Question 3 *Hacked EvanBot* ()

Hacked EvanBot is running code to violate students' privacy, and it's up to you to disable it before it's too late!

```
 1 #include <std
 2
 3 void spy_on_s
 4     char buff
 5     fread(buf                n);
 6 }
 7
 8 int main() {
 9     spy_on_students();
10     return 0;
11 }
```

The shutdown code for Hacked EvanBot is located at address `0xdeadbeef`, but there's just one problem—Bot has learned a new memory safety defense. Before returning from a function, it will check that its saved return address (rip) is not `0xdeadbeef`, and throw an error if the rip is `0xdeadbeef`.

Clarification during exam: Assume little-endian x86 for all questions.

Assume all x86 instructions are 8 bytes long. Assume all compiler optimizations and buffer overflow defenses are disabled.

The address of `buffer` is `0xbffff110`.

Q3.1 (3 points) In the next 3 subparts, you'll supply a malicious input to the `fread` call at line 5 that causes the program to execute instructions at `0xdeadbeef`, *without* overwriting the rip with the value `0xdeadbeef`.

The first part of your input should be a single assembly instruction. What is the instruction? x86 pseudocode or a brief description of what the instruction should do (5 words max) is fine.

Q3.2 (3 points) The second part of your input should be some garbage bytes. How many garbage bytes do you need to write?

○ (G) 0 ○ (H) 4 ○ (I) 8 ○ (J) 12 ○ (K) 16 ○ (L) ——

Q3.3 (3 points) What are the last 4 bytes of your input? Write your answer in Project 1 Python syntax, e.g. `\x12\x34\x56\x78`.

Q3.4 (3 points) When does your exploit start executing instructions at `0xdeadbeef`?

○ (G) Immediately when the program starts

○ (H) When the `main` function returns

○ (I) When th [illegible] unction returns

○ (J) When th [illegible] rns

○ (K) ——

○ (L) ——