CS161 Spring 2026 — Introduction to Computer Security — Discussion 2

Q1 *Software Vulnerabilities* (4 points)

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

This code contains several security vulnerabilities. **Circle *three* such vulnerabilities** in the code and briefly explain each of the three on the next 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;
}
```

Q1.1 (1 point) Explanation:

Solution: Line **15** has a fencepost error: the conditional test should be `i < n` rather than `i <= n`. The test at line **14** 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 **20** will write past the end of `kittens`, representing a buffer overflow vulnerability.

Q1.2 (1 point) Explanation:

Solution: At line **15**, 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.

Q1.3 (1 point) Explanation:

Solution: On line **18** 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.

Alternate Solution: On line **24** 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.

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 **24**.

Q1.4 (1 point) Describe how an attacker could exploit these vulnerabilities to run shellcode:

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 arbitrary code.

Q2 *Hacked EvanBot* (12 points)

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 <stdio.h>
 2
 3  void spy_on_students(void) {
 4    char buffer[16];
 5    fread(buffer, 1, 24, stdin);
 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`.

Q2.1 (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.

```
jmp *0xdeadbeef
```

Solution: You can't overwrite the rip with `0xdeadbeef`, but you can still overwrite the rip to point at arbitrary instructions located somewhere else. The idea here is to overwrite the rip to execute instructions in the buffer, and write a single jump instruction that starts executing code at `0xdeadbeef`.

Q2.2 (3 points) The second part of your input should be some garbage bytes. How many garbage bytes do you need to write?

○ 0 ○ 4 ○ 8 ● 12 ○ 16

Solution: After the 8-byte instruction from the previous part, we need another 8 bytes to fill buffer, and then another 4 bytes to overwrite the sfp, for a total of 12 garbage bytes.

Q2.3 (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`.

`\x10\xf1\xff\xbf`

Solution: This is the address of the jump instruction at the beginning of `buffer`.

Q2.4 (3 points) When does your exploit start executing instructions at `0xdeadbeef`?

○ Immediately when the program starts

○ When the main function returns

● When the `spy_on_students` function returns

○ When the `fread` function returns

Solution: The exploit overwrites the rip of `spy_on_students`, so when the `spy_on_students` function returns, the program will jump to the overwritten rip and start executing arbitrary instructions.

Q3 *I Understood that Reference!* (12 points)

Consider the following vulnerable C code:

```
void vulnerable(int start, char *ptr) {
    ptr[start] = ptr[3];
    ptr[start + 1] = ptr[2];
    ptr[start + 2] = ptr[1];
    ptr[start + 3] = ptr[0];
}

void helper(int8_t num) {
    if (num > 124) {
        return;
    }
    char arr[128];
    fgets(arr, 128, stdin);
    vulnerable(num, arr);
}

int main(void) {
    int y;
    fread(&y, sizeof(int), 1, stdin);
    helper(y);
    return 0;
}
```

Assume that:

- You are on a little-endian 32-bit x86 system.
- There is no other compile padding or saved additional registers.

Write your answer in Python 3 syntax (just like in Project 1).

Q3.1 (2 points) Fill in the stack diagram below, assuming that execution has entered the call to `vulnerable`:

Stack

RIP of `main`
SFP of `main`
`y`
`num`
RIP of `helper`
SFP of `helper`
`arr`
`ptr`
`start`
RIP of `vulnerable`
SFP of `vulnerable`

Solution: Notice that when integer arguments are passed to functions, their values are directly placed on the stack (not pointers, like strings).

For the rest of this question, assume that the RIP of `main` is located at `0xbfffdc0c` and that your malicious shellcode is located at `0xef302010`.

In the next two subparts, construct an exploit that executes your malicious shellcode.

Q3.2 (5 points) Provide an input to the variable `y` in the `fread` in `main`.

For this subpart only, you may write a decimal number instead of its byte representation.

−12

Solution: This attack involves noticing that we're indexing into the `ptr` array using a value that we control (we choose the value of `start` through the `fread` call in `main`). With this, we can think about how to overwrite one of the RIP's present on our stack. There's a catch, though — since `start` is restricted to values less than 127, and `arr` is 128 bytes long, we can't write over the RIP of `helper`; however, we can set `start` to a negative number to index downwards and override the RIP of `vulnerable`. That RIP lives three words below the start of the array, so we start at array index `-12`.

Any number with the final byte set to `'\xf4'` will work. We want to choose some `y` such that, when cast to an `int8_t`, it becomes `-12`.

Q3.3 (5 points) Provide an input to the variable `arr` in the `fgets` in `helper`.

`\xef\x30\x20\x10`

Solution: We need to reverse the order of the bytes in our new RIP address, since they're read in reverse of our normal direction (starting at `ptr[3]` and going to `ptr[0]`). Once this address is placed into the array, it'll be in little-endian format.