Peyrin Kao
Fall 2022

CS 161 Computer Security

Discussion 2

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`.

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

```c
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:

Solution: Line **12** has a fencepost error: the conditional test should be $i < n$ rather than $i <= n$. The test at line **11** 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 **17** will write past the end of **kittens**, representing a buffer overflow vulnerability.

2. Explanation:

Solution: At line **12**, 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 **15** 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.

Solution: Another possible solution is that on line **21** 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 **21**.

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 arbitrary code.

Question 2 ***Echo, Echo, Echo*** **()**

Consider the following vulnerable C code:

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

char name[32];

void echo(void) {
    char echo_str[16];
    printf("What do you want me to echo back?\n");
    gets(echo_str);
    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. For the first 4 parts, assume that **no memory safety defenses** are enabled.

Q2.1 (2 min) Assume that execution has reached line 8. Fill in the following stack diagram. Assume that each row represents 4 bytes.

Stack
1
2
RIP of `echo`
SFP of `echo`
3
4

○ (A) (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

○ (B) (1) - SFP of `main`; (2) - RIP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

● (C) (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[12]`; (4) - `echo_str[8]`

○ (D) ——

○ (E) ——

○ (F) ——

Solution: The first two items on the stack are the RIP and SFP of `main`, respectively. Since the stack grows down, lower addresses are at the bottom of the diagram, and arrays are filled from lower addresses to higher addresses and are zero-indexed. As such, row (3) contains `echo_str[12]`, and row (4) contains `echo_str[8]`.

Q2.2 (3 min) Using GDB, you find that the address of the RIP of `echo` is `0x9ff61fc4`.

Construct an input to `gets` that would cause the program to execute malicious shellcode. Write your answer in Python syntax (like in Project 1). You may reference `SHELLCODE` as a 16-byte shellcode.

Solution: Where to put the `SHELLCODE` does not matter. This is a simple stack-smashing attack: we want to redirect execution to `SHELLCODE` when the `echo` function returns.

Approach 1: Place the Shellcode in the Buffer

```
SHELLCODE + 'A' * 4 + '\xb0\x1f\xf6\x9f'
```

Approach 2: Place the Shellcode above the RIP

```
'A' * 20 + '\xc8\x1f\xf6\x9f' + SHELLCODE
```

There may be a few other correct answers here (with the shellcode placed at slightly different offsets within the buffer or above the RIP), but these are the most common.

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 <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. [1]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.

> **Solution:** jmp *0xdeadbeef
>
> 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`.
>
> Grading: most likely all or nothing, with some leniency as long as you mention something about jumping to address `0xdeadbeef`. We will consider alternate solutions, though.

[1]In practice, x86 instructions are variable-length.

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) ——

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.

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`.

Solution: `\x10\xf1\xff\xbf`

This is the address of the jump instruction at the beginning of `buffer`. (The address may be slightly different on randomized versions of this exam.)

Partial credit for writing the address backwards.

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 the `spy_on_students` function returns

○ (J) When the `fread` function returns

○ (K) ——

○ (L) ——

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.