Weaver
Spring 2022

CS 161
Computer Security

Discussion 1

For a handy guide on x86 and GDB, check out this GDB Cheatsheet.

Question 1 ***Stack Diagram Practice*** ()

Consider the following function.

```
1  void swap(int* num1, int* num2, int arr_local[]) {
2      int temp = *num1;
3      *num1 = *num2;
4      arr_local[0] = *num1;
5      *num2 = temp;
6      arr_local[1] = *num2;
7  }
8
9  int main(void) {
10     int x = 61;
11     int y = 1;
12     int arr[2];
13     swap(&x, &y, arr);
14 }
```

1. Complete the diagram of the stack if the code were executed until a breakpoint set on 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, and each box represents 4 bytes in memory. The bottom of the page represents the lower addresses.

main's sfp
x
y
arr
arr
int* arr_local
int* num2
int* num1
swap's rip
swap's sfp
temp

2. Now, draw arrows on the stack diagram denoting where the ESP and EBP would point if the code were executed until a breakpoint set on line 3.

Question 2 ***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.

```
1  struct cat {
2      char name[64];
3      char owner[64];
4      int age;
5  };
6
7  /* 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. */
8  size_t search_basket(struct cat *basket, int n, char *owner_name) {
9      struct cat kittens[32];
10     size_t num_kittens = 0;
11     if (n > 32) return -1;
12     for (size_t i = 0; i <= n; i++) {
13         if (basket[i].age < 12) {
14             /* Reassign the owner name. */
15             strcpy(basket[i].owner, owner_name);
16             /* Copy the kitten from the basket. */
17             kittens[num_kittens] = basket[i];
18             num_kittens++;
19             /* Print helpful message. */
20             printf("Adopting kitten: ");
21             printf(basket[i].name);
22             printf("\n");
23         }
24     }
25     /* Adopt kittens. */
26     adopt_kittens(kittens, num_kittens); // Implementation not shown
           .
27     return num_kittens;
28 }
```

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 3 ***Remus*** ()

This problem is Question 1 of Project 1 converted to a discussion question, with the intention of providing a foundation for completing Project 1. The question will also consist of a live GDB walkthrough, conducted by your TA. A video version of this walkthrough is available on https://cs161.org/.

Consider the following vulnerable code.

```
1  #include <stdio.h>
2
3  void orbit()
4  {
5   char buf[8];
6   gets(buf);
7  }
8
9  int main()
10 {
11  orbit();
12  return 0;
13 }
```

1. Which line of code contains the memory safety vulnerability? Briefly explain this vulnerability.

   Solution: Line **6** contains buffer overflow vulnerability with `gets()`. There are no checks that the `stdin` input to `gets()` is less than or equal to the length of `buf`, or 8 bytes. Thus, an attacker can use `gets()` to perform a buffer overflow attack, overwriting the RIP to point to an attacker's shellcode.

2. Complete the stack diagram if the code were executed until a breakpoint set on line 6. 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, and each box represents 4 bytes in memory. The bottom of the page represents the lower addresses.

orbit's RIP
orbit's SFP
compiler padding
compiler padding
buf
buf

3. Construct an input to `buf` that would result in a successful buffer overflow attack. Assume that orbit's `RIP` is stored at `0xffffff8e0` and that you have a `SHELLCODE` script that you would like to execute. In addition, assume that there are 8 bytes of compiler padding.

Solution: `'A' * 20 + '\xe4\xf8\xff\xff' + SHELLCODE`

With this input, we overwrite orbit's RIP to point to the SHELLCODE stored above the RIP.