Homework Assignment 3

● Graded

Student

HARRY KIM

Total Points

100 / 100 pts

Question 1

Summarization of vulnerabilities

100 / 100 pts

Identifying the stack buffer overflow

✔ **+ 20 pts** Following the template to clearly and correctly explain the vulnerability

+ 15 pts Following the template to explain the vulnerability but incomplete/inaccurate information is provided

+ 10 pts Following the template to explain the vulnerability but the analysis does not make sense

---

Identifying the heap buffer overflow

✔ **+ 20 pts** Following the template to clearly and correctly explain the vulnerability

+ 15 pts Following the template to explain the vulnerability but incomplete/inaccurate information is provided

+ 10 pts Following the template to explain the vulnerability but the analysis does not make sense

---

Identifying the Use After Free

✔ **+ 20 pts** Following the template to clearly and correctly explain the vulnerability

+ 15 pts Following the template to explain the vulnerability but incomplete/inaccurate information is provided

+ 10 pts Following the template to explain the vulnerability but the analysis does not make sense

---

Identifying the integer overflow

✔ **+ 20 pts** Following the template to clearly and correctly explain the vulnerability

+ 15 pts Following the template to explain the vulnerability but incomplete/inaccurate information is provided

+ 10 pts Following the template to explain the vulnerability but the analysis does not make sense

---

Identifying the use of uninitialized variables

✔ **+ 20 pts** Following the template to clearly and correctly explain the vulnerability

+ 15 pts Following the template to explain the vulnerability but incomplete/inaccurate information is provided

+ 10 pts Following the template to explain the vulnerability but the analysis does not make sense

Question assigned to the following page: 1

Harry Kim
u1226472

CS 4440 Assignment 3

Stack Buffer Overflow/91.c

```
10  void CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cat_01_bad()
11  {
12      char * data;
13      char dataBadBuffer[50];
14      char dataGoodBuffer[100];
15
16      data = dataBadBuffer;
17      data[0] = '\0';
18      {
19          char source[100];
20          memset(source, 'C', 100-1);
21          source[100-1] = '\0';
22
23          strcat(data, source);
24          printLine(data);
25      }
26  }
```

From the above code, we can see a stack buffer overflow caused by the statement “strcat(data, source)” at line 23. The function “strcat” appends the string pointed to by “source” to the end of the string pointed to by “data”. We can see that “source” is initialized and has all of its data except the last element set to the character ‘C’ and at the last element, “source[100-1] = ‘\0’” on lines 19 through 21. Since “data” holds only one character ‘/0’, appending the data from “source” to the end of “data”, “data” will have more characters than it can hold (as it points to “dataBadBuffer” on line 16) and thus “strcat(data, source)” will cause a stack overflow.

Question assigned to the following page: 1

Harry Kim
u1226472

Heap Buffer Overflow/67.c

```
13  static void badSink()
14  {
15      char * data = CWE122_Heap_Based_Buffer_Overflow__c_CWE805_char_memcpy_45_badData;
16      {
17          char source[100];
18          memset(source, 'C', 100-1);
19          source[100-1] = '\0';
20
21          memcpy(data, source, 100*sizeof(char));
22          data[100-1] = '\0';
23          printLine(data);
24          free(data);
25      }
26  }
27
28  void CWE122_Heap_Based_Buffer_Overflow__c_CWE805_char_memcpy_45_bad()
29  {
30      char * data;
31      data = NULL;
32
33      data = (char *)malloc(50*sizeof(char));
34      if (data == NULL) {exit(-1);}
35      data[0] = '\0';
36      CWE122_Heap_Based_Buffer_Overflow__c_CWE805_char_memcpy_45_badData = data;
37      badSink();
38  }
```

From the above code, we can see that the heap buffer overflow is caused by “memcpy(data, source, 100*sizeof(char))” on line 21. Similar to a stack buffer overflow, this code overflows the max amount of space allocated for “data” where it leaks over by copying the character ‘C’ more times than the max size of “data” and overwrites some data in the allocated memory which in turn causes an overflow.

Question assigned to the following page: 1

Harry Kim
u1226472

Use After Free/20.c

```
10  void CWE416_Use_After_Free__malloc_free_int64_t_02_bad()
11  {
12      int64_t * data;
13
14      data = NULL;
15      if(1)
16      {
17          data = (int64_t *)malloc(100*sizeof(int64_t));
18          if (data == NULL) {exit(-1);}
19          {
20              size_t i;
21              for(i = 0; i < 100; i++)
22              {
23                  data[i] = 5LL;
24              }
25          }
26
27          free(data);
28      }
29      if(1)
30      {
31
32          printLongLongLine(data[0]);
33
34      }
35  }
```

From the above code, we can see a use-after-free caused by the statement at line 32. We can see that “data” is allocated on line 17 and is filled with data from lines 21 to 24. On line 27, data is freed which leaves the pointer created on line 12 as a dangling pointer, but after “data” is freed, it is accessed and used again on line 32. Thus, a use-after-free happens.

Question assigned to the following page: 1

Harry Kim
u1226472

Integer Overflow/78.c

```
12  void CWE190_Integer_Overflow__unsigned_int_max_postinc_04_bad()
13  {
14      unsigned int data;
15      data = 0;
16      if(STATIC_CONST_TRUE)
17      {
18
19          data = UINT_MAX;
20      }
21      if(STATIC_CONST_TRUE)
22      {
23          {
24
25              data++;
26              unsigned int result = data;
27              printUnsignedLine(result);
28          }
29      }
30  }
```

From the above code, the integer overflow happens on line 25. We can see that “data” is set to the max integer value on line 19. When “data” is overflowed past that max limit, the integer in “data” is looped back around and set to 0 which is what “result” set to. Thus, an integer overflow has occurred.

Question assigned to the following page: 1

Harry Kim
u1226472

Use of Uninitialized Varable/73.c

```
8   void CWE457_Use_of_Uninitialized_Variable__struct_array_malloc_no_init_01_bad()
9   {
10      twoIntsStruct * data;
11      data = (twoIntsStruct *)malloc(10*sizeof(twoIntsStruct));
12      if (data == NULL) {exit(-1);}
13
14      ;
15
16      {
17          int i;
18          for(i=0; i<10; i++)
19          {
20              printIntLine(data[i].intOne);
21              printIntLine(data[i].intTwo);
22          }
23      }
24  }
```

From the above code, the use of uninitialized variables is demonstrated by lines 20 and 21. We can see that “data” is never really initialized as it is only allocated for space on line 11. Since “data” is never initialized, printing the contents on lines 20 and 21 causes it to print some completely different data which can be exploited to be used maliciously. Thus, a use of an uninitialized variable has occurred.