CS2107 Tutorial 8 (Software Security)

School of Computing, NUS

April 15, 2021

1. *(Format string & buffer overflow vulnerabilities)*: Try out this `badprogram.c` C program:

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

int main(int argc, char **argv)
{
    char text[16];

    strcpy(text, argv[1]); /* copy the 1st argument into array "text" */
    printf("This is how you print correctly:\n");
    printf("%s", text);
    printf("\n");

    printf("This is how not to print:\n");
    printf(text);
    printf("\n");

    return 0;
}
```

Compile the program above (e.g. `gcc -o badprogram badprogram.c`), and execute it. Notice that the program takes in an argument. By executing, for instance, `./badprogram 'hello world'`, the program `badprogram` will take in the string `'hello world'` as an argument (without the two quote characters), store the argument into the array `text`, and then print it.

(a) When the input is `'hello world'`, how many characters will be copied to the array `text` by `strcpy`? 11 or 12?

Solution

12. There is a null character.

Try running it with different arguments. What would happen if the argument is:

(a) `two words`

(b) `'two words'`

(c) `'%d %d'`

(d) `%4p`

(e) `%s`

(f) `'%s %s %s'`

(g) `'helloworld%n'`

In (e), the process likely would crash with the error message: `segmentation fault`. Explain what has happened.

Solution

The OS detects and raises "segmentation fault" when a process attempts to access (include both read and write) a memory location that the process does not has the right to access. In this case, the printf attempts to print out the content of a string. To do so, it needs to *read* the content of the string. In C, string is represented as a pointer. So, the printf process will treat the value stored in the supposingly 2nd parameter as a pointer, and then read the location pointed by the pointer. With good chance, the pointer points to location that would could "segmentation fault".

Solution

You can just try running the executable with the stated arguments. The error message varies across different compilers. The output lines are (note: the character '`/`' below is used to separate the two output lines for each given argument):

(a) `two / two`

(b) `two words / two words`

(c) `%d %d /` ⟨*some_integer*⟩ ⟨*some_integer*⟩

(d) `%4p /` ⟨*some_pointer_value*⟩

(e) `%s /` ⟨*some_string*⟩ or segmentation fault

(f) `%s %s %s /` ⟨*some_string*⟩ but very likely Segmentation fault.

(g) ⟨*the_entered_long_string*⟩ `/ *** stack smashing detected ***: ./badprogram terminated Aborted (core dumped)`

2. Consider this program.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
```

```
{
    char * text;
    unsigned char A; /* A is infront of t.  B is behind t */
    char t[5];
    unsigned char B;

    text = t;  A=0;  B=0;

    strcpy(text, argv[1]); /* copy the 1st argument into array "text" */
    printf ("the string is: %s\n", text);
    A='A';       B='B';
    printf ("the string is: %s\n", text);
}
```

What would happen if the argument is

(a) `1234`

(b) `12345`

Max allocation is only 5, so 5 characters will remove the \0 null char to stop the string

Solution

One of the assignments `A='A'` or `B='B'` actually overwritten the location `text[6]` which is the null character. So, the character right after `12345` is not null. Hence, printf will continue printing until it hits the null character.

3. *(UNIX commands)*: Familiarize yourself with Unix/Linux commands like `ls`, `cat`, `sh`, `echo`.

Solution

Please refer to your UNIX/Linux references, and do your own practice. A free and good resource to learning Linux commands is: William Shotts, "The Linux Command Line", `http://linuxcommand.org`.

4. *(Safe/unsafe functions)*: Find out more about the following C library functions. Which usages should be avoided, and why?

(a) `strcat (dest, source);`

(b) `strncat (dest, source, n);`

(c) `memcpy (dest, source, n);`

(d) `strncpy (dest, source, strlen(source));`

(e) `printf (f, str);`

(f) `printf ("hello my name is %s", str);`

(g) `sprintf (str, f);`

(h) `printf ("Please key in your name:  "); gets (str);`

(i) `scanf ("%s", str);`

(j) `scanf ("%20s", str);`

Solution

(a) Unsafe. The buffer `dest` can get overflowed since there is no limit to the number of characters concatenated into it.

(b) Safe if `n` $\leq$ the remaining characters (bytes) available on the `dest` buffer.

(c) Safe if `n` $\leq$ the size of the `dest` buffer.

(d) Unsafe. The buffer `dest` can get overflowed since `strlen(source)` can be greater than `dest`'s length.

(e) Potentially unsafe if `f` comes from a user input.

(f) Safe.

(g) Unsafe. The buffer `str` can get overflowed since the formatted string output can be longer than `str`'s length.

(h) Unsafe. The buffer `str` can get overflowed since there is no limit to the number of characters read and stored into it.

(i) Unsafe. The buffer `str` can get overflowed since there is no limit to the number of characters read and stored into it.

(j) Safe if the size of `str` $\geq$ 20 since at most 20 characters are stored by `scanf()` into `str`.

5. *(Memory initialization)*: Consider the following C program.

```
#include <stdio.h>

int main()
{
  unsigned char a[10000];

  for (int i=0; i<10000; i++)
      printf ("%c", a[i]);

  return 0;
}
```

(a) What would be the output? What is its implication to secure programming?

(b) A possible preventive measure is to always initialize the array. What is the disadvantage of doing that?

Solution

(a) In C, the value of an uninitialized local variable is *indeterminate/undefined*, which basically can be anything. Accessing such an uninitialized variable leads to an "*undefined behavior*". The program above, which prints out the uninitialized array `a`, poses a security risk since the printed data could be sensitive. This is the case since memory chunks in a running process' memory are basically "recycled", both between different process executions and during the process execution. If the array `a` happens to contain a sensitive piece of information, the information will then get leaked out to the external user.

(b) Extra processing time is required.

6. *(Integer overflow)*: Consider the following C program.

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

int main()
{
   unsigned char a, total, secret; // Each of them is a 8-bit unsigned integer
   unsigned char str[256];       //  str is an array of size 256
   a = 40;
   total = 0;
   secret = 11;

   printf ("Enter your name: ");
   scanf ("%255s", str);         // Read in a string of at most 255 characters

   total = a + strlen(str);

   if (total < 40) printf ("This is what the attacker wants to see: %d\n", secret);
   if (total >= 40) printf ("The attacker doesn't want to see this line.\n");
}
```

If the user follows the instruction and enters his/her name honestly, he/she will be unable to see the secret. Suppose you are the attacker, how would you cause the secret number to be displayed?

Solution

Notice that `total` is a 8-bit unsigned integer. Hence, its value ranges from 0 to 255. Addition operations done on `total` are thus actually carried out under module 256.
To display the secret number `secret`, we need to overflow `total`, yet make its overflowed value less than 40, i.e. between 0 and 39 (inclusive). To this end, we need to supply a string with length between $256-40 = 216$ and 255 (inclusive).

Notice, however, that the secret number still be displayed if we enter a string longer than 255. *Why?* (Hint: pay attention to the format specifier supplied to the `scanf()` invocation in the code.)

7. *Terminologies*: CVE, Black list, White list, Black hat, White hat, Spamhaus, CERT, SingCert, SOC (Security Operation Center), SIEM.

Solution

Please Google/Wiki the terminologies.

— End of Tutorial —